UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE ESTRATEGIAS SELF-X EN UNA ARQUITECTURA DE CONTROL COGNITIVO ARTIFICIAL

(ISCOSI_87/1314)

Carmelo Juanes Rodríguez
Tutor: Rodolfo Haber
Ponente: Pablo Varona

JULIO 2014

Abstract

Micromanufacturing processes are complex physical processes which are continuously changing in a dynamic environment. In this context the main objective is the development of computational technologies and algorithms in order to enable faster, self-organized, self-optimized behavior of micromanufacturing processes by means of intelligent control systems. In this work we design and implement an artificial cognitive architecture for controlling manufacturing processes. Moreover, hardware aspects are also considered in order to deploy the developed architecture on low-cost platform hardware in order to reduce the cost of the hosts.

This work is based on a methodology with six main steps. Firstly, we have designed an artificial cognitive architecture, inspired in the Modified Shared Circuits Model approach, using UML. After that, the architecture is developed and implemented in Java. Once the architecture is developed, we have built an instantiation of the architecture in order to test it by simulation studies and actual experiments in an industrial setup. Other targets of this work are the instantiation with an off-line optimization algorithm based on *cross-entropy* method and an on-line learning algorithm based on *Q-learning* method. Thereby we have built an instantiation with self-optimization and self-learning capabilities. It is important to note that inference models are fuzzy inference systems and Adaptive Neural Fuzzy Inference Systems (ANFIS) models written in C/C++, invoked from Java by means of the *SWIG* technology.

In addition, we have provide the instantiation with the feature of running in a distributed environment on the basis of *ZeroC Ice* middleware. Thus the instantiation will run in two Raspberry Pi, with the cognitive unit and the executive unit deployed in each low-cost computing platform. We have also used the *IceGrid* service to provide the instantiation with the ability of discovering host without having to enter manually their IP addresses.

From the best of our knowledge, one of the main contributions of this work is not only the design and implementation of the artificial cognitive control architecture but also to assess the performance using simulation studies and real time tests in an industrial setup of micromanufacturing plan. The goal of theses experiments is to check the suitability of the functions implemented in the architecture and to demonstrate its control capability running in a low-cost hardware.

Relying on the experimental results we have demonstrated that the artificial cognitive control yields good results and promising opportunities to deal with complex systems even running in a low power machine as the Raspberry Pi. But most important is that this work give us the basement and the computational framework to enable new functions in order to improve the developed artificial cognitive architecture.

Keywords: artificial cognitive architecture, self-optimization, learning, micromanufacturing.

Resumen

Los procesos de microfabricación son complejos procesos físicos que están continuamente cambiando en un entorno dinámico. En este contexto, el principal objetivo es el desarrollo de tecnologías computacionales y algoritmos con el fin de permitir un comportamiento de los procesos de microfabricación más rápido, auto organizado y auto optimizado por medio de sistemas de control inteligente. En este trabajo diseñamos e implementamos una arquitectura cognitiva artificial para controlar los procesos de fabricación. Además, los aspectos del hardware se han tenido en cuenta con el fin de desplegar la arquitectura desarrollada en una plataforma hardware de bajo coste para reducir los gastos de los equipos.

Este trabajo está basado en una metodología con seis pasos principales. Primero, hemos diseñado una arquitectura cognitiva artificial inspirada en la aproximación Modified Shared Circuits Model, usando UML. Después de esto, la arquitectura ha sido desarrollada en Java. Una vez que la arquitectura ha sido implementada la hemos instanciado con el fin de probarla a través de estudios de simulación y experimentos reales en una instalación industrial. Otros objetivos de este trabajo son la instanciación con un algoritmo offline de optimización basado en el método de *entropía cruzada* y un algoritmo online de aprendizaje basado en *Q-learning*. De este modo hemos construido una instaciación con capacidades de auto optimización y auto aprendizaje. Es importante destacar el uso de sistemas de inferencia borrosa y sistemas de inferencia borrosa neuronal adaptados (ANFIS) escritos en C/C++, invocados desde Java gracias a la tecnología *SWIG*.

Además, hemos dotado a la instanciación de la capacidad de ejecutarse en un entorno distribuido basado en el middleware *ZeroC Ice*. En consecuencia, la instanciación se ejecutará en dos Raspberry Pi, con la unidad cognitiva y la ejecutiva desplegadas en cada una de ellas. También hemos usado el servicio *IceGrid* para dotar a la instanciación de la habilidad de descubrir los equipos sin tener que introducir manualmente las direcciones IP.

Desde nuestra experiencia, la principal contribución de este trabajo es, además del diseño y la implementación de la arquitectura cognitiva artificial, la evaluación del rendimiento usando estudios de simulación y pruebas en tiempo real en una instalación industrial de microfabricación. El objetivo de estos experimentos es comprobar la idoneidad de las funciones implementadas en la arquitectura y demostrar su capacidad de control ejecutándose en una hardware de bajo coste.

Confiando en los resultados experimentales hemos demostrado que el control cognitivo artificial produce buenos resultados y promete oportunidades para tratar con sistemas complejos incluso ejecutándose en máquinas de poca potencia. Pero más importante es que este trabajo nos ofrece una base y un marco de trabajo computacional que habilitará nuevas funciones para mejorar la arquitectura cognitiva artificial que hemos desarrollado.

Palabras clave: arquitectura cognitiva artificial, aprendizaje, auto-optimización, microfabricación.

Acknowledgements

A research project like this is never the work of anyone alone. I would like to express my sincere gratitude to the people who contributed in different ways to the achievement of this assignment.

A special thanks goes out to my university classmates for their affection, which have turned these five years into an unforgivable experience.

I would like to express the deepest appreciation to Alberto Gimeno Sánchez, who helped me a lot revising my English and providing me all the assistance I needed even when he had more important things to do.

I also must acknowledge Jose Ramón Elbal Sánchez, who advised me with the graphical issues of the project trying to make it light and easy to read.

My cousin, for making her best to do the last revision of the English in this work so as to avoid all possible mistakes.

My girlfriend, for supporting me whenever I needed and always raising a smile even in the worst moments.

More concretely, I acknowledge the research groups, C4LIFE and GAMHE, which supported me. Specially, Francisco Penedo Álvarez for his help with the design and implementation phases, Fernando Castaño Romero who made all we needed to realize the experiments of our work, even when unexpected problems appeared, and Dr. Rodolfo Haber Guerra who has supported me from time ago and has done his best effort to inspect my work and make this work possible.

Last but not least, I want to offer my sincerest gratitude to my parents, to my father since he always was a reference to me and this has encouraged me hugely to achieve my objectives, and to my mother, for the immeasurable efforts she has done to make possible my reaching at this point of my career, sincerely, thanks.

Contents

Appendix

List of Figures

List of Tables

List of Algorithms

Acronyms

A | C | D | G | I | J | M | P | R | S | U | X

A

ANFIS
Adaptive Neural Fuzzy Inference Systems. I

C

CE
Cross entropy. 7, 16, 17, 33, 42, 49

CORBA
Common Object Request Broker Architecture. 8–10

D

DDE
Dynamic Data Exchange. 34

G

GUI
Graphical User Interface. 72

I

IDE
Integrated development environment. 23

IDL
Interface Definition Language. 9, 22

J

JNI
Java Native Interface. 24

JVM
Java Virtual Machine. 24

M

MDP
Markov decision process. 8, 19

MSCM
Modified Shared Circuits Model. I, III, 11, 12, 14, 34, 36

MSE
Mean Squared Error. 18, 21, 36

P

PDF
Probability Density Functions. 15, 16

R

RMI
Remote Method Invocation. 10

RTSJ
Real-Time Specification for Java. 23

S

SCM
Shared Circuits Model. 5, 6, 11, 12

SWIG
Simplified Wrapper and Interface Generator. 24, 33, 34, 49

U

UDP
User datagram protocol. 8

UML
Unified Modeling Language. I, III, 23, 24

X

XML
Extensible Markup Language. 24

1 Introduction

In this section we will explain the motivations that encouraged us to carry out this TfG. Then we will describe the framework which make this work possible. In addition, the main and the specific objectives of the TfG will be roughly defined. Finally, the overall structure of the document will be presented.

1.1 Motivation

The main rationale of this work and the significant roots are motivated in one of the most important objectives since years ago is to emulate the human brain, or at least some of its capabilities: response to a stimulus, learning or mirroring. Recent results in different disciplines, such as neuroscience, psychology, artificial intelligence and results related with new machines and intelligent processes, have laid on the computational theory of intelligence [1]. There are many definitions of intelligence; one of them is the ability of human beings to perform new, highly complex, unknown or arbitrary cognitive tasks efficiently and to explain those tasks with brief instructions. Based on this concept many researchers have explored new paradigms to achieve a qualitative change and then create new artificial cognitive control strategies.

A natural cognitive system displays effective behavior through perception, action, deliberation, communication and both individual interaction and interaction with the environment. What makes a natural cognitive system different is that it can function efficiently under circumstances that were not explicitly specified when the system was designed, in other words, these systems have certain flexibility for dealing with the unexpected [2]. In addition, a cognitive system can learn from experience to improve how it operates, it can be aware of its own behavior and reflects on its own capabilities, and it can respond robustly to unexpected changes.

On the other hand, manufacturing is a clear example of a dynamic technical system operating in an environment characterized by continuous changes. In these scenarios the main objective is the development of technologies and algorithms that enable faster, self-organized, self-optimized behavior process control systems. Additionally, manufacturing processes are conditioned by the presence of nonlinear and time-variant dynamics that are determined by forces, torques and other variables. These characteristics increase the functional complexity of manufacturing and the functional and precision requirements of sensors, actuators and computing resources.

Due to all discussed above, we propose an artificial cognitive architecture that enables the control of manufacturing processes implementing some of the capabilities that define the human brain.

1.2 Framework

This TFG is supported by the national research project *DPI2012-35504: Artificial Cognitive Control System for Micromanufacturing Processes. Method and Application (CONMICRO)* led by Prof. Rodolfo Haber. This project has provided the necessary umbrella to carry out the required tasks during this period.

Nowadays, as we have said previously, there is a great interest, some work done, and on-going research on cognitive architectures for control, especially in the field of robotics. The main objective of CONMICRO is to design and implement a cognitive architecture for controlling micromanufacturing processes, specifically the micro-drilling and micro-milling processes. The scientific basis relies on the nexus between the paradigm of internal model control and brain-cerebellum connectivity as a basis of human intelligence, and a cognitive architecture based on imitation of own abilities and socio-cognitive experiences was then proposed.

CONMICRO aims to go beyond the current state of the art through research in four main pillars, namely: a stochastic optimization method based on hybridization with a view to self-optimization, a coordination mechanism by managing goals based on Artificial Intelligence techniques, a strategy of reinforcement learning to provide the architecture for learning capabilities and a computationally efficient strategy (based on AI techniques) of switching between modules that form the cognitive architecture. Thus, we intend to continue evolving artificial cognitive controllers with advanced neurofuzzy and fuzzy strategies that lead to appropriate local and global representation of micro-scale complex processes.

The main singularity of CONMICRO is the biological inspiration supported by the most recent results of Computer Science and Artificial Intelligence in conjunction with strategies and methods of Control Engineering and Automation. Therefore, the cognitive architecture is linked to control laws, strategies and performance indices of control engineering (e.g., internal model control) bringing an alternative scheme to perform efficiently control tasks, similar to how humans perform certain cognitive processes. While conventional control architectures focused on solving problems with a hierarchy of goals and sub-goals, little has been done about how to achieve the same level of skills in cognitive-inspired architecture as intended by CONMICRO.

The expected scientific contributions of CONMICRO are related to the technical contributions by means of a method that will support the practical feasibility of artificial cognitive control strategies for micromachining processes. Mechanical micromachining processes are manufacturing processes that are of great importance and complexity but the performance of current control strategies is inadequate and insufficient, or in many cases nonexistent. The developed method will be applied to micro scale processes (micro-

milling and micro-drilling) in which control of micro-forces and micro-torque is essential. In short, the method of designing a cognitive control system will provide the what, how and why of goal-oriented behaviour in cooperating individual modules. Therefore, CONMICRO is focused on converging techniques in the field of COGNO-INFO-MICRO, according to the arguments above presented.

1.3 Objectives

Accordingly to our framework this TfG is focused on the four main pillars of the CONMICRO project. Obviously, for the sake of time constraints, we will choose three of them to work on. Thereby, the general objectives of this work are:

1. To design an artificial cognitive architecture for controlling physical processes based on the results of the COGNETCON project. This architecture will be based on the shared circuits model of sociocognitive capacities. It is important to notice this architecture will be designed regarding the principles of simplicity and scalability.

2. To implement the designed artificial cognitive architecture for controlling physical processes. This implementation will be done in Java for reasons that we will describe later in Section 3. The implementation has also to take into account simplicity and scalability principles as its main issue.

3. To validate the design and implementation of the artificial cognitive architecture in both simulation and a real scenario. Due to the importance of the machine in the real scenario it is essential that the application matches perfectly with the simulation study.

4. To acquire new knowledge and developing skills in different fields and topics of the real world mainly: working with proprietary software and programming languages, for instance Labview; using middleware for enable communication between different machines such ZeroC Ice; the design and implementation of the artificial cognitive architecture, to learn about modeling and control in real time of physical process.

We can highlight some specific objectives:

1. Accordingly with the first pillar of the CONMICRO project a stochastic optimization method will be developed. The method chosen to be developed is the well-known cross entropy algorithm.

2. To design and develop a strategy of reinforcement learning in correspondence with other of the CONMICRO's pillar. This method will be based on the reinforcement learning Q-learning algorithm with some modification to work on-line.

3. To embed the implementation of the artificial cognitive architecture in a low-cost hardware platform, e.g., Raspberry Pi.

4. To acquire knowledge and skills in relation with proprietary software of the main manufacturers in the industrial environment such as Siemens, National Instruments, etc.

5. To gain knowledge and experience in the development of applications built on a middleware, in our case ZeroC Ice.

1.4 Structure of the document

In the following sections we will present the main steps in order to accomplish this work. In section 2 we will present the state of the art describing the current state of the artificial cognitive architectures and their integration with the present self capabilities, namely, self organization and self optimization; as well as the value of building it over a middleware for enabling artificial cognitive control in a remote manner.

Furthermore, in section 3, we will expose the tools and technologies that we have used to build this project. Firstly, we will explain the direct antecedent of our cognitive architecture, the shared circuits model of sociocognitive capacities, and how we have improved some of its features. Secondly, we will describe deeply the middleware technology that we have chosen, ZeroC Ice. In addition, we will argue about the importance of the low-cost hardware and how they can help us in our objective of designing low-cost cognitive control architecture. Finally, we will focus on the description of the tools we have used to model our artificial cognitive architecture.

In section 4 we will focus on the design and development of the architecture. Firstly, we will cover the requirement analysis to understand better the design that we will choose; then, the global design of the architecture will be exposed. To conclude this section two cases of use will be presented and explained.

Once we have detailedly explained the design of the artificial cognitive architecture and the cases of use where it has been tested, we are prepared to show the results of these tests. These results will be presented with graphical charts for a better understanding. These points will be extensively described in section 5.

Finally, we will present our Conclusions in section 6. Additionally to our conclusions we will describe the lines of work we will focus on in the near future.

2 State of the art

In this section we are going to present an analysis of the main concepts and technologies in which our work is based. Firstly, we will analyze the basis of our work with the main antecedent of the Shared circuits model and the architecture built with that theory. Then, we will make a brief introduction to the self-capabilities, focusing on self-optimization and self-organization. Finally, we will present the limitations of the developed algorithms, in terms of accessibility and range, when we want to deploy them in a factory and how a middleware can take care of the communication between hosts enabling us to center only in the logic of our algorithms.

2.1 Cognitive architectures for controlling physical processes

The antecedent of the artificial cognitive architecture which we have based on is the Shared Circuits Model (SCM) approach [3]. SCM approach is supported on a layered structure to describe how certain human capacities (i.e., imitation, deliberation, and mindreading) can be deployed thanks to subpersonal mechanisms of control, mirroring, and simulation. Basically, SCM is based on the observation of the human brain. Some brain regions are in charge of coding actions for reaching objectives and how other regions code means for reaching objectives. So that, the brain may be envisaged as making use of not only inverse models that estimate the necessary motor plan for accomplishing an objective in a given context, but also a forward model that enables the brain to anticipate the perceivable effects of its motor plan, with the object of improving response efficiency. The first kind of behavior is covered by the action of the layer 1 of SCM, while the behavior described in the forward model is covered by layer 2 of SCM. Layer 4 of the scheme is the layer in charge of controlling when to perform one type of behavior or another.

Other kind of behavior is the imitation that, in addition to playing an important role in both the sociability and the development of the human adult, is a means of learning. Imitative learning requires mirroring the actions of others in response to given circumstances. In order to perform this task, first the observer copies the input/output associations already observed, inhibiting the mirroring mechanism. SCM represents this mirroring capacity in its layer 3. The interaction between layer 3 and the inhibition control performed by layer 4 serves to emulate the agent's capacity to distinguish self from other.

SCM also describes, from a functional point of view, how the agent can carry out the cognitive skill of mindreading. This capacity is emulated by the operation of layer 5, which is in charge of simulating possible other-related inputs that are external (exogenous) to the agent.

A graphical summary of this approach can be seen in figure 1.

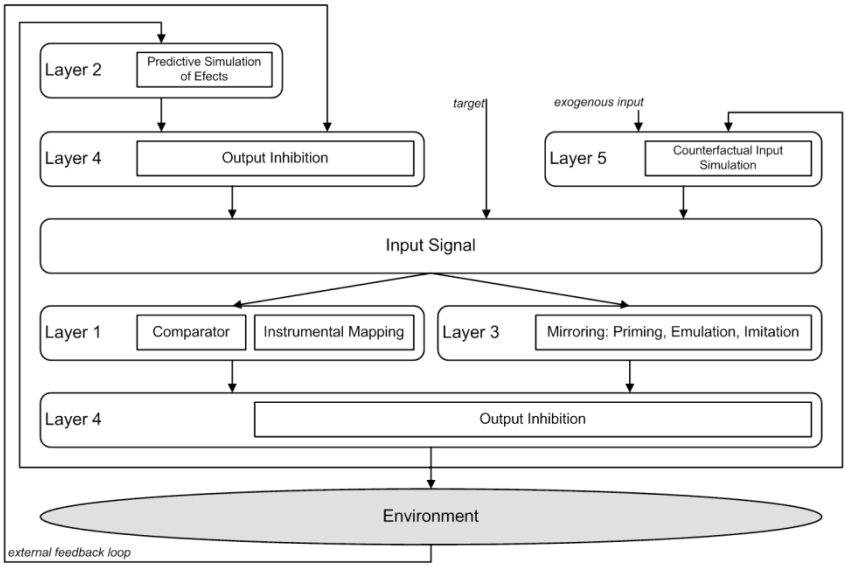

Figure 1: Description of SCM from [3].

Based on this approach is our main antecedent [4]. In that work some modifications of the SCM approach are done for making possible a first design and implementation of a control architecture to a high-performance drilling process. In next section we will focus on the details of that work.

2.2 Self capabilities

The term selfware has been referred to as a growing set of self-properties that are emerging in the Autonomic Computing and other related self-managing systems initiatives. Natural self-organizing systems work without central control and operate based on contextual local interactions. The particularity of self-organized systems is the capacity to spontaneously (without external control) produce a new organization in case of environmental changes.

During the 60's the concept of self-organization was adopted by the scientific community in the field of System Theory, and later in the 70's and 80's it became a common place to those advocated to the research of complex systems. Its application to the study of complex systems have been widely explored and significant efforts have been made, and are still being made, by the scientific community to find new approaches and apply those ones inspired in nature and social systems in order to elucidate the basic principles of self-organization theory and practice. Thereafter, and mainly in the last decade, it has been an abundance of scientific literature on self-organization, but as the subject is still the object of intense research and development, different interpretations of the concepts can be found. Although a lack of consistency is normal in an evolving science, it does not

contribute to uniform global understanding.

Another aspect related with self capabilities is how to emulate human skills in artificial cognitive architecture. Nowadays, there is a wide range of approaches from self-adaptivity up to self-resilience. In this work, two main self-X capabilities will be considered: self-optimization and self-learning. These self-X capabilities can bring important characteristic to the artificial cognitive architecture by enabling adaptation before noise, disturbances and nonlinearities with less human interaction and reducing manual reconfiguration procedures. In our discussion, we will briefly introduce self-optimization and self-learning. A thorough analysis on both fields goes beyond the scope of this TfG.

Nowadays there is a huge body of literature on deterministic and stochastic methods for solving optimization problems. It is beyond the scope of this TfG to analyze and compare all the available gradient free optimization methods for the optimal tuning of control systems. These methods encompass a wide range of techniques ranging from genetic algorithms to particle swarm optimization [5, 6]. This TfG focuses on a well-established stochastic method that is emerging in the field of optimal tuning. The Cross entropy (CE) method is inspired by an adaptive variance minimization algorithm for estimating probabilities of rare events for stochastic networks [7]. The main rationale of CE is the construction of a random sequence of solutions which converges probabilistically to the optimal or near-optimal solution using two iterative stages [8]. First a sample of random data (e.g., a set of controller parameters) is generated according to a specified random mechanism. A better sample is produced in the next iteration, updating the parameters of the random mechanisms (i.e. parameters of the probabilistic density functions) using the corresponding data [9].

Reinforcement learning is a learning paradigm with learning by reward/ penalty with some interesting applications in controlling complex systems so as to maximize a numerical performance measure that expresses a long-term objective. What distinguishes reinforcement learning from supervised learning is that only partial feedback is given to the learner about the learner's predictions. Further, the predictions may have long term effects through influencing the future state of the controlled system. Thus, time plays a special role. The goal in reinforcement learning is to develop efficient learning algorithms, as well as to understand the algorithm' merits and limitations. Reinforcement learning is of great interest because of the large number of practical applications that it is able to address, ranging from problems in artificial intelligence to operations research or control engineering.

There are several works reported in the literature in which reinforcement learning has an important function in control systems, such as tuning fuzzy PD and PI controllers [10], force control of an industrial robot [11] or trajectory control in robots [12]. The analysis of all available reinforcement learning methods goes beyond the scope. For a fairly comprehensive catalog

of learning problems with a description of an important number of state of the art algorithm see [13].

In our work we center on a well-known algorithm called Q-learning. The main rationale of this choice is the simplicity of the approach, its model-free feature and the results reported in the literature based on this algorithm. Mainly Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It performs by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. Additionally, Q-learning can handle problems with stochastic transitions and rewards, without requiring any adaptations.

The main advantage of this algorithm is that, once it is implemented, we can focus on building a good reward function that matches with our specific problem. As we will explain later in the next section, we have to modify this algorithm to adapt it to our "special state space" and to execute it on-line.

2.3 Middleware for enabling artificial cognitive control

Nowadays the scientific community is working to bridge the gap between laboratory and factory by improving the way sensors are connected to and integrated with other devices. One key issue in the endeavor is the long-distance monitoring and control of complex plants, which requires synergy strategies that ally smart devices and communication technologies with advanced computational methods. For such an alliance to exist, sensor interfacing must be improved, and one way of doing that is to develop remote monitoring systems based on self-powered systems and classical communication methods, such as infrared and LCD [14]. On the other hand, the problem of interconnecting sensors or transducers in distributed process monitoring can also be addressed by analyzing the feasibility of an Internet-based interface. A low-cost, smart, Internet sensor-based User datagram protocol (UDP) with Internet visibility is an alternative solution that requires the use of proprietary protocols and the direct management of PC ports [15]. Yet another possibility is to use distributed object-computing middleware, which enables common network programming tasks to be automated regardless of considerations such as what communication protocols and networks are used to interconnect distributed objects. This is the solution we have chosen in our work.

The first option that we have to analyze is the Common Object Request Broker Architecture (CORBA). CORBA technology provides a clear opportunity for process monitoring and strategic process control (i.e., optimization of complex systems based on performance indices) in complex electromechanical systems, as it provides an adequate trade-off in terms of

middleware performance, resource consumption and available technical support at the early design stage. CORBA uses its Interface Definition Language (IDL) to define interfaces with server objects. CORBA has intrinsic location transparency, because clients can invoke servers without worrying about the physical location of the server objects. Moreover, CORBA's interoperability enables communication via TCP/IP-based Internet Inter-ORB protocol (IIOP) regardless of the platforms and operating systems of the clients and servers involved.

In real-time CORBA (RT-CORBA) specification, mechanisms and policies are defined to control processor resources, communication resources and memory resources to support the real-time distributed requirements of the application fields [16].

The TAO real-time ORB, the Adaptive Communication Environment (ACE) ORB, unlike most CORBA implementations on the market (MT-Orbix, CORBAplus, Visiobroker, miniCOOL, Orbacus, Java IDL), provides, among other advantages, predictable behavior which is an important feature in real time application. The TAO real-time ORB core shares a minimum part of ORB resources, thus substantially reducing synchronization costs and the priority inversion between the process threads. Furthermore, TAO is a freeware real-time CORBA implementation with open source code, built within the framework of components and patterns provided by ACE.

The second option among current middleware is ZeroC Ice. Ice provides a communication solution that is simple to understand and easy to deal with. Yet, despite its simplicity, Ice is flexible enough to accommodate even the most demanding and mission-critical applications.

Ice allows you to write your distributed applications in C++, Java, C#, Python, Ruby, PHP, and ActionScript. With Ice Touch, your application can include Objective-C components that run on the iPhone, iPad, and iPod touch, while Ice for Java can also be used to build Ice applications for Android. All these features, at the time of this writing, give this middleware an important added value.

Ice was designed from the ground up for applications that require the utmost in performance and scalability. At the network level, Ice uses an efficient binary protocol that minimizes bandwidth consumption. Ice uses little CPU and memory, and its highly efficient internal data structures do not impose arbitrary size limitations. This allows applications to scale to tens of thousands of clients with ease. By their creators: data can be transmitted at whatever speed is supported by the network, so Ice does not create any performance bottleneck. A comparison with other popular distributed computing solutions can be found here [17].

One of the most important things behind Ice is its rich set of services, such as event distribution, firewall traversal with authentication and filtering, automatic persistence, automatic application deployment and monitoring, and automatic software distribution and patching. All services can be

replicated for fault tolerance, so they do not introduce a single point of failure. The use of these services greatly reduces development time because they eliminate the need to create distribution infrastructure as part of application development.

Other distributed computing solution that has to be studied when Java is the programming language chosen to out project, is Remote Method Invocation (RMI). Java RMI enables the programmer to create distributed Java technology-based to Java technology-based applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

RMI focuses on Java, with connectivity to existing systems using native methods. This means RMI can take a natural, direct, and fully-powered approach to provide you with a distributed computing technology that lets you add Java functionality throughout your system in an incremental, yet seamless way.

Although the design and application of artificial cognitive control architecture based on middleware is essential because the middleware facilitates communication between different hosts, the design and development of the architecture is done in a middleware-free manner namely, independent of the middleware chosen to enable communication.

We have chosen for our implementation ZeroC Ice. The main reasons are the good results shown in [17], the reasonable advantages against CORBA [18] and its versatility and ease of use.

3 Tools and technologies

In the following pages we will describe the technologies which our work is based on and the tools that we will use to achieve our goal.

3.1 Underlying technologies

We are going to explain the main applied technologies and how we have adapted theses technologies to this particularly work.

3.1.1 Modified shared circuits model

The basement of this TfG is the project CONMICRO and one of the result of this project is the work [4] where the Modified Shared Circuits Model (MSCM) is proposed.

Five modules are constructed made up of one or more processes performed by the layers of Shared Circuits Model (SCM). The MSCM proposal defines each module in terms of cognitive ability that emulates. MSCM embodies a computational infrastructure that is plausible from a neuroscience and psychological viewpoint. For the sake of a better understand of this architecture a brief description of each module follows.

Module 1: basic adaptive feedback control
: From the perspective on System Theory and Computational Science, module 1 of MSCM is equivalent to layer 1 in SCM. Module 1 is represented by a controller C and an optimization/adjustment process for this controller. This controller performs the instrumental association between input and output, similar to the description in SCM and very similar to closed-loop control systems. Similar to layer 1 of SCM, the inputs are a reference signal r , according with the objectives, and the system output y''. The control signal u'' is the output of this module.

Module A: performance index computation
: This module estimates a performance index or figure of merit. This is an essential module because the performance index that is calculated by this module takes part in the decision about when modules act and in what modules combined or not run. A performance index J is calculated by weighting the figures of merit J_i selected by selector J_s according with the actual objectives and goals.

Module 0: objective management
: The main role of module 0 is to supply a set of reference signals that module 1 uses to achieve the eventual objectives. MSCM can handle multi-objectives by technical, production, economic, and other objectives into references r_i and the corresponding figures of merit J_i at the system's executive level.

Module 2: simulative prediction of effects for improved control
In module 2 there is a set of forward models M on whose basis, given a control signal, a future output is generated. In order to perform this task, it is also relevant to take into account the actual characteristics of the environment, i.e., the exogenous input and/or the influence of noise and disturbances.

Module 2 runs before the action is performed in order to evaluate/deliberate about different action possibilities, depending on whether the agent's criteria are successful or not. However, it is always functioning with the purpose of comparing its output with current process output, thus module 1 can learn.

Module 3: mirroring for priming, emulation and imitation
MSCM proposal enriches SCM work addressing "learned knowledge" as that knowledge that has been incorporated into the set of instrumental associations, that module 1 handles in MSCM. In the meantime, the knowledge that expresses imitation, used in layer 3 in SCM, remains in the set of managed by module 3 in MSCM.

Therefore, on the asis of error signal e the output generates action, mirroring the behavior of others. The input of module 3 similarly to the input for module 1 may be determined either by the system input plus feedback or by the effects simulated by module 2.

Module 4: management of monitored output inhibition
MSCM module 4 is very close to SCM layer 4. Module 4 performs executive functions within the system, namely make the decision whether to enact deliberation (through the operation of module 2) or imitation (which is enabled in module 3). To make this decision module 4 uses the performance index J provided by module A.

Module 5: counterfactual input simulation
This module is in charge of simulating effects while running off-line. In order to perform this task, there is a set of forward models, D, that gather the associations between dictated action u'' and the possible, or counterfactual, inputs. In MSCM module 4 is the module that decides when to run this simulation.

The integration between modules is summarized in figure 2.

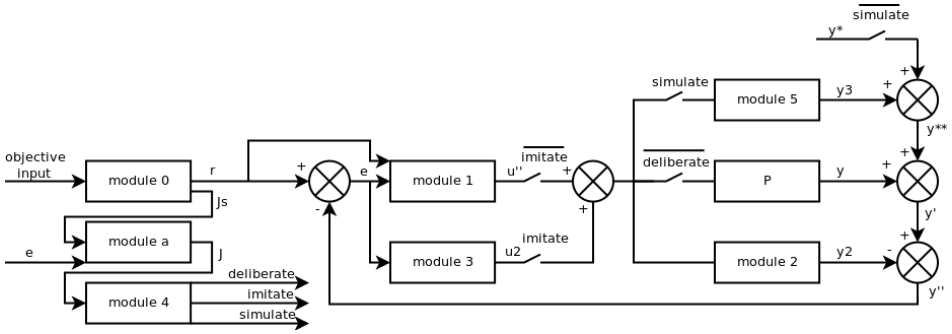

Figure 2: Expanded block diagram of MSCM from [4].

As we have aforementioned, module 4 is in charge of governing the action, imitation and learning cycle. The algorithm that module 4 follows is presented in algorithm 1.

Algorithm 1: Algorithm of the system's action, imitation, and learning cycle.

1 Deliberate the possible actions to carry out, by means of interaction of modules 1, 2 and 5.
2 If any of the actions leads to success, execute it.
a) If there is noise in excess, module 2 learns the new effects that have been produced.
3 If not,
a) If noise surpasses a threshold, go to (3.b.iii.1).
b) If not,
. i) Deliberate about the possible actions of others (imitative actions), through the interaction of modules 2, 3 and 5.
. ii) If any of the imitative actions leads to success, execute it.
. (1) Learn this actions by incorporating the corresponding instrumental association into module 1's private set of forward models.
. iii) If not,
. (1) Through an optimization process, acquire a new action using the process model handled by module 2, whose results are handled by said optimization process in module 1.
. (2) Execute new action by means of the operation of this module.

Particularly, in our work, we are going to implement a simplified version of this architecture, focusing on three special configuration of the blocks of MSCM. The first configuration is composed exclusively of module 1 of MSCM, due to the nature of module 1, we are going to call this configuration *Single loop* (figure 3).

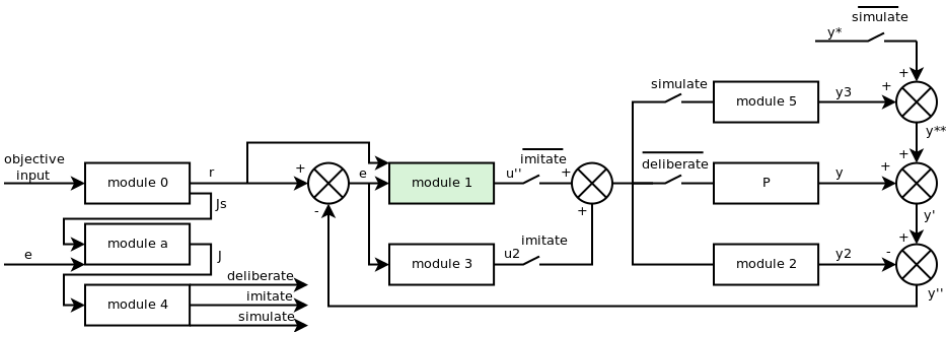

Figure 3: Diagram of configuration *Single loop*.

The second configuration is composed essentially of module 3. This module stores inverse models so, activating only this module, we achieve a configuration called *Anticipation* (figure 4).

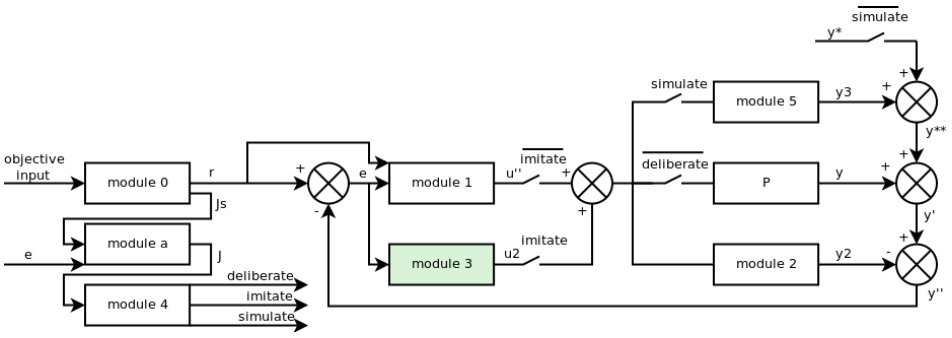

Figure 4: Diagram of configuration *Anticipation*.

The last configuration is called *Anticipation + Mirroring*. Its name comes from the activation of module 2 and 3 enabling such mirroring skill (figure 5).

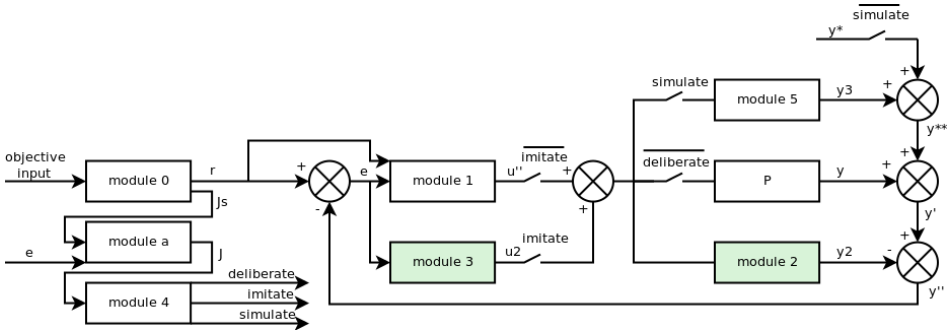

Figure 5: Diagram of configuration *Anticipation + Mirroring.*

3.1.2 Cross entropy method for self-optimization

The cross-entropy method was first introduced to estimate the probabilities of events with very small probabilities. Later it was adapted for optimizing systems on the assumption that sampling around the optimum of a function has a very small probability.

In the optimization of control parameters $K_1, \cdots, K_n$ on the basis of a suitable criterion, the control system is considered to be a stochastic system. Therefore, the cross-entropy method is applied as a population based optimization algorithm in the sense that it utilizes the scores of the trial runs as optimal in a rhetorical sense.

Let X be a random variable on a space χ, P_X its Probability Density Functions (PDF) and let the score ϕ be a real function in χ. The cross-entropy method aims to find the minimum of ϕ over χ and the corresponding states x' satisfying this minimum:

$$\gamma' = \phi(x') = \min_{x \in \chi} \phi(X) \tag{1}$$

The cross entropy method provides a methodology for creating a sequence of $x_0, x_1, \cdots, x_N$ and levels $\gamma_0, \gamma_1, \cdots, \gamma_N$ such that $\{\gamma\}_N$ converges to γ' and $\{x\}_N$ converges to x'.

We are concerned with estimating the probability $l(\gamma)$ of an event $E_\gamma = \{x \in \chi | \phi(x) \geq \gamma\}$, $\gamma \in \mathbb{R}^+$.

Defining a collection of functions for $x \in \chi, \gamma \in \mathbb{R}^+$

$$I_\gamma(x, \gamma) = I_{\{\phi(x_i) \geq \gamma\}} = \begin{cases} 1 & if \quad \phi(x) \leq \gamma \\ 0 & if \quad \phi(x) > \gamma \end{cases} \tag{2}$$

Let $g(\cdot, v)$ be a family of probability density functions on χ parameterized by a real valued vector $\vec{v} \in \mathbb{R}^+$ and $g(x, v) | v \in \Gamma$.

$$l(\gamma) = P_v(\phi(x) \geq \gamma) = E_v \cdot I_v(x, v) \tag{3}$$

where E_v denotes the corresponding expectation operator.

In this manner equation (3) converts the optimization problem into an associated stochastic problem with very small probability using a variance minimization technique such as *importance sampling* where the random sample is drawn from a *priori* appropriate PDF, h. Taking a random sample $x_0, x_1, \cdots, x_N$ from an importance sampling (different) density h on χ and evaluating:

$$\hat{l} = \frac{1}{N} \sum_{i=1}^{N} I_{\{\phi(x_i) \geq \gamma\}} \cdot W(x_i) \tag{4}$$

where $\hat{l}$ is called the importance sampling and $W(x) = \frac{g(x,v)}{h(x)}$ is called the likelihood ratio.

Searching the optimal sampling density $h'(x)$ is problematic, since determination of $(h'(x)$ requires l to be known.

$$h'(x) = \frac{I_{\{\phi(x_i) \geq \gamma\}} \cdot g(x, v)}{l} \tag{5}$$

Thus the parameter vector, called the referenced parameter or tilting parameter v', should be chosen such that the distance between h' and $g(x, v)$ is minimal, reducing the problem to a scalar case.

A measure of distance between two densities g and h is the Kullback-Leibler distance, also called *cross-entropy* between g and h:

$$D(g, h) = \int g(x) \cdot \ln g(x) dx - \int g(x) \cdot \ln h(x) dx \tag{6}$$

Minimizing $D(g(x, v), h')$ is equivalent to maximizing $\int h'(x) \cdot \ln g(x, v) dx$ which implies:

$$\max_v D(v) = \max_v E_p(I_{\{\phi(x_i) \geq \gamma\}} \cdot \ln g(x, v)) \tag{7}$$

Using again the importance sampling, we can rewrite (7) to compute the expectation in (7). Therefore we can draw a sample $x_1, x_2, \cdots, x_N$ from g and estimate the maximum (or minimum) of $\hat{D}(V)$:

$$\max_v \hat{D}(v) = \max \frac{1}{N} \sum_{i=1}^{N} I_{\{\phi(x_i) \geq \gamma\}} \cdot \frac{P_x(x_i)}{h(x_i)} \cdot \ln g(x_i, v) \tag{8}$$

However h is still unknown in (7). The CE algorithm tries to overcome this difficulty by adaptively constructing a sequence of parameters $(\gamma_i | t \geq 1)$ and $(v_i | t \geq 1)$.

For the sake of simplicity and background of the research groups C4LIFE from EPS (UAM) and GAMHE from CSIC, in this TfG we have applied a simple and flexible strategy of using a population based optimization method

for control systems on the basis of simulation. The algorithm generates a set of controller parameters and calculates the cost function or performance index: the CE algorithm is then applied considering two main stages. The first one consists of the generation of random samples using $g(x, v)$ and the calculation of a performance index or cost function. The second one is to update $g(x, v)$ using data collected in the first stage via the CE method.

The main CE adapted to solve optimization problems is as follows. Consider the optimization problem:

$$\phi(x') = \gamma' = \max_{x \in \chi} \phi(x) \tag{9}$$

The rationale behind CE for optimization is to convert the problem (9) into an associated stochastic problem and then solve it adaptively as the simulation of a rare event. If γ' is the optimum of ϕ, the main issue is to define a family $g(\cdot, v)|v \in \Gamma$ and iterate the CE algorithm such as $\gamma_i \to \gamma'$ to draw samples around the optimum.

The algorithm can be summarized as follows:

Algorithm 2: Cross-entropy algorithm

1 Calculate $v_t = v_0$

2 Generate a sample of size $N(x_i^t)_{1 \leq i \leq N}$ from $g(x, v_t)$, compute $\phi(x_i^t)$, and order $\phi_1 \geq \phi_2 \geq \cdots \geq \phi_N$ from biggest to smallest. Use $\gamma_t = \phi_{[\rho N]}$ to select the elite subset of population.

3 Update v_t with:

$$v_{t+1} = arg \min_{v} \frac{1}{N} \sum_{i=1}^{N} I_{\{\phi(x_i^t) \geq \gamma_t\}} \cdot \ln g(x_i^t, v_t) \tag{10}$$

4 Repeat step 2 until convergence or ending criterion.

5 Assume that convergence is reached at $t = t'$, an optimal value for ϕ can be obtained from $g(\cdot, v_t')$.

The parameter updating step 3 is performed using the best performance samples $N^{elite} = \rho N$, also called elite samples. The updated parameters are found to be maximal likelihood estimates (MLEs) of the elite samples [19]. The sampling distribution can be quite arbitrary, and does not need to be related to the function that is being optimized [9]. The normal (Gaussian) distribution function gives easy and simple updating formula.

The mean $\vec{\mu}$ and variance $\vec{\sigma}$ are estimated for each iteration t considering $j = 1, \cdots, n$ parameters (e.g. if the controller has two parameters, the input scaling factors (K_e, K_{ce}), then $n = 2$) as:

$$\mu_j = \sum_{i=1}^{N^{elite}} \frac{x_{ij}}{N^{elite}} \tag{11}$$

$$\sigma_j = \sum_{i=1}^{N^{elite}} \frac{(x_{ij} - \mu_j)^2}{N^{elite} - 1} \tag{12}$$

where $4 \leq N^{elite} \leq 20$.

The mean vector $\vec{\hat{\mu}}$ should converge to γ' and the vector of standard deviation $\vec{\hat{\sigma}}$ should converge to the zero vector. A smoothing parameter α for the mean vector and dynamic smoothing β for the variance are applied in order to prevent the occurrences of $0s$ and $1s$ in the parameter vectors.

$$\vec{\hat{\mu}} = \alpha \cdot \vec{\hat{\mu}} + (1 - \alpha)\vec{\hat{\mu}} \tag{13}$$

$$\vec{\hat{\sigma}} = \beta \cdot \vec{\hat{\sigma}} + (1 - \beta)\vec{\hat{\sigma}} \tag{14}$$

$$\beta_t = \beta - \beta(1 - \frac{1}{1t})^q \tag{15}$$

where $0.4 \leq \alpha \leq 0.9$, $0.6 \leq \beta \leq 0.9$ and $2 \leq q \leq 7$.

The performance index that we considered in the optimization problem (9) is the Mean Squared Error (MSE).

$$MSE = \frac{1}{N} \sum_{i=1}^{N} e_i(t)^2 \tag{16}$$

where $e_i(t) = \frac{ref_i - y_i}{ref_i}$, i.e., $e_i(t)$ is the relative error between the reference value and the process output. Therefore, the performance index is really a relative MSE.

3.1.3 Q-learning method for self-learning

The problem model, the MDP, consist of an agent, states S and a set of actions per state A. By performing an action $a \in A$ the agent can move from state to state. Executing an action in a specific state provides the agent with a reward. The goal of the agent is to maximize its total reward. It does this by learning which is the best action for each state. Therefore the algorithm has a function which calculates the *Quality* of a state-action combination, $Q : S \times A \to \mathbb{R}$.

Before learning has started, Q can return any fixed value, chosen by the designer of the problem. Then, each time the agent selects an action, receives its rewards and enters in the new state. The core of the algorithm is a simple value iteration update. It assumes the old value and makes a correction based on the new information:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(R_{t+1} + \gamma \cdot \max_{a \in A} Q_t(s_{t+1}, a) - Q_t(s_t, a_t)) \quad (17)$$

where s_t is the state in time t, a_t is the action taken in time t, R_{t+1} is the reward received after performing action a_t, α_t is the learning rate and γ is the discount factor which trades off the importance of sooner versus later rewards.

Normally Q-learning is executed in a episodic manner where an episode ends when state s_{t+1} is a final state. However, Q-learning can also learn in non-episodic tasks.

It can be noted that Q-learning does not specify a method for select the action to perform in each state. However there are several policies to select an action, e.g., the well-known ϵ-greedy or *softmax* policies.

A summary of this algorithm can be found in algorithm 3.

Algorithm 3: Q-learning algorithm

```
1  Initialize Q(s_t, a_t) arbitrary (or with a fixed value);
2  Initialize s_0 to one arbitrary state (or one fixed);
3  foreach episode do
4      repeat
5          foreach step episode do
6              Choose a_t using the adequate policy;
7              Perform action a_t. Receive R and s_{t+1};
8              Update Q-values with equation (17);
9              s_t ← s_{t+1};
10         end
11     until s_t is a terminal state ;
12 end
```

In the following lines we are going to present the modifications that we have done to the original algorithm to make it compatible with our architecture.

The main tuning action is more convenient than necessary but helps to simplify our implementation of Q-learning. In our architecture the learning is performed to find optimal parameters of determined models of our architecture. We assume that a state in our architecture is a set of parameters of the model/models that are learning, thus we identify each state unequivocally with a set of parameters:

$$s_t \leftrightarrow (K_1^{(t)}, K_2^{(t)}, \cdots, K_N^{(t)})$$

So the actions to change to one state to another are those that change at least one parameter of the set $(K_1^{(t)}, K_2^{(t)}, \cdots, K_N^{(t)})$ in this manner our Q-values functions is $Q(s_t, a_t) = Q(s_t)$.

In our work we discretize the continuous space of the variables for simplicity as already reported in [10]. Thus each parameters K_i has its own limits (K_i^{min}, K_i^{max}) determined by the model the parameters belongs to. Then if we would use M possible values of each parameter, the range of values of this parameter would be:

$$K_{i_1} = K_i^{min}, K_{i_2} = K_{i_1} + \frac{K_i^{max} - K_i^{min}}{M-1}, \cdots, K_{i_M} = K_i^{max} \tag{18}$$

As a consequence of the exposed above, our space of states is finite whose dimension is M^N. Due to the restrictions of the real environment we can not make long step in a specific parameter in a given moment of time. Thus our actions will be limited to achieve this restriction.

For a given state $s_t \leftrightarrow (K_1^{(t)}, K_2^{(t)}, \cdots, K_N^{(t)})$ its available actions will be those that change s_t to $s_{t+1} \leftrightarrow (K_1^{(t+1)}, K_2^{(t+1)}, \cdots, K_N^{(t+1)})$ where:

$$K_i^{(t+1)} \in [\max(K_i^{min}, K_i^{(t)} + step), \min(K_i^{max}, K_i^{(t)} + step)]$$

namely in each action only one *step* in each parameter can be done.

In our architecture, the learning mechanism will run at slower frequency than the control mechanism because, in order to perform a correct learning, the process has to be run for a sufficient time, which resemblance cascade concept or a hierarchical approach with different bandwidths. Taking into account this factor if our control mechanism has a period of time between control action of $p_{control}$, the learning has to be performed at least ten times more, it is, $p_{learning} = \delta p_{control}$, where $\delta \in \mathbb{Z}, \delta \geq 10$. With this, our reward function is defined:

$$R = \begin{cases} +500 & if \quad \phi(t) \leq 0.05 \\ +100 & if \quad 0.05 < \phi(t) \leq 0.1 \\ -100 \cdot \phi(t) & if \quad 0.1 < \phi(t) \end{cases} \tag{19}$$

where $\phi(t)$ is the performance index associated with the taken action and has the following expression:

$$\phi(t) = \frac{1}{\delta}\sum_{i=1}^{\delta}(\frac{ref_i - y_i^{(t)}}{ref_i})^2 \tag{20}$$

where ref_i is the reference value in time $t + i \cdot p_{control}$, $y_i^{(t)}$ is the process's output in time $t + i \cdot p_{control}$ with the parameter set $(K_1^{(t)}, K_2^{(t)}, \cdots, K_N^{(t)})$. As we can see $\phi(t)$ is the MSE evaluated in $[t, t + \delta \cdot p_{control}]$.

With all these changes, the function to update the $Q - values$ has to change to:

$$Q_{t+1}(s_{t+1}) = Q_t(s_{t+1}) + \alpha_t(R_{t+1} + \gamma \cdot \max_{a \in A} Q_t(s_{t+2}) - Q_t(s_{t+1})) \tag{21}$$

For the sake of simplicity in this first approach we will use the ϵ-greedy policy for choose action because in almost all scenarios it is sufficient. The ϵ-greedy policy follows the algorithm 4.

Algorithm 4: ϵ-greedy policy algorithm.

```
r = random();
if r < ε then
    Take a random action between all possible actions.
else
    Take the action with produce the state with more Q-value.
end
```

To summarize all the steps that we have described the algorithm 5 is presented.

Algorithm 5: Modified Q-learning algorithm

Initialize $Q(s_t)$ arbitrary (or with a fixed value obtained with some method as we will explain later);
Initialize s_0 to one arbitrary state (or one fixed);
repeat
  foreach *step* **do**
    Choose a_t using the ϵ-greedy policy;
    Perform action a_t and change to state s_{t+1};
    Wait $\delta \cdot p_{control}$ and receive R;
    Update Q-values with equation (21);
    $s_t \leftarrow s_{t+1}$;
  end
until s_t *is a terminal state* ;

3.1.4 Zero-C Ice middleware

As we have said, we will use Zero-C Ice to distribute our architecture. Like some other middlewares Ice has a IDL to describe the program you want to build. For our tests we only make a few components of our architecture distributed. Due to this choice, we can describe four distributed units:

Cognitive unit
: This unit contains all the *cognitive* components: learning and optimization mechanism, organization logic, the execution logic and the application itself. It is expected this unit will be deployed in a low-cost computational hardware.

Executive unit
: This unit contains the models that our architecture will use: single loop, direct, inverse and simulation models. It is expected this unit will be deployed in a low-cost computational hardware.

Process unit
: The process itself has to be distributed because is one of the objective: remote control of a physical process.

With this description we will write an Ice specification file and parse it with the program:

```
slice2java
```

this program will generate several auxiliary class that ZeroC Ice needs to make the communication through the net. As we will see we have to extends some of the generated classes in order to give some functionality to these generic classes.

In addition to the common use of Ice, we will use one of its service, namely *IceGrid*. Mainly we will use this service because IceGrid enables clients to discover their servers. By acting as an intermediary, IceGrid decouples clients from their servers and helps to improve the performance and reliability of applications with its support for replication, load balancing and automatic fail over. As we will see later, some configuration files has to be created to set the service.

The program needed to execute the IceGrid registry process is the program `icegridregistry`. It is given with the installation of Ice, as well as the program needed to run a server, `icegridnode`.

3.1.5 Raspberry Pi: low-cost computing platform

The Raspberry Pi is a low cost, credit-card sized computer that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It is capable of doing everything you would expect a desktop computer to do, from browsing the Internet and playing high-definition video, to making spreadsheets, word-processing, and playing games.

The Raspberry Pi has the ability to interact with the outside world, and has been used in a wide array of digital maker projects, from gaming machines [20] to open-source voice computing [21]. We want to contribute in both manufacturing system and Raspberry Pi environment deploying part of our architecture in this hardware to show that important tasks needed to control different processes can be perform in a low-cost hardware.

3.2 Tools for modeling and implementation

The first thing that we have to do in order to create our artificial cognitive architecture is design it, so for the design of the artificial cognitive architecture we will use the well-known Unified Modeling Language (UML) [22]. UML is a general-purpose modeling language in the field of software engineering, which is designed to provided a standard way to visualize the design of a system.

The programming language that we have chosen is Java because of its spread in the real world, its ease of use and its facilities to work in different environments with too much work to add. In addition, the Real-Time Specification for Java (RTSJ) provides an advantage when we want to extend our work to a full real time environment [23].

Once we have the design of our application (both class diagram and use case diagram) we can start to implement our architecture. For this purpose we will use an Integrated development environment (IDE), the popular *Eclipse*. We have chosen work with an IDE because it helps a lot in order

to work with Java. Furthermore, if we do the design with the Eclipse UML plug-in *Papyrus*, Eclipse can generate a lot of code from it.

Since we have to work with different platforms and operating system we want to automatically perform some of the actions that we have to constantly repeat. In order to accomplish this task, we have chosen the tool *Ant*. Ant is a tool that allows us to start programmed tasks from a description file written in Extensible Markup Language (XML). Ant works similar to the well-known *Makefile*.

Finally, in order to take advantage of the previous work of our research group, we will use Simplified Wrapper and Interface Generator (SWIG) [24]. SWIG is an interface compiler that connects programs written in C and C++ with several languages such as Perl, Python or Java. It works by taking the declarations found in C/C++ header files and using them to generate the wrapper code that languages need to access the underlying C/C++ code. In addition, SWIG provides a variety of customization features that let you tailor the wrapping process to suit your application. The task that SWIG realizes is possible thanks to the Java Native Interface (JNI) framework which enables the communication between the Java Virtual Machine (JVM) and programs written in C, C++ or assembly.

With SWIG we can use the controllers written in C/C++ for testing our application. Since we can not find an ant task of SWIG we will create a *Makefile* to automatically execute the process of running SWIG.

4 Design and development

In this section we will present the design of the artificial cognitive architecture together with the requirement analysis. Then we will explain the main details of our implementation without entering in deep explanations.

Once we will have finished with the details of our artificial cognitive architecture we will present an implementation of the architecture. We can see this implementation as a particular instantiation of our architecture which we will use in our experiments.

Finally, two experiments will be presented. The first result is based on a simulation environment with no real physical machine. The second experiment is carried out in a real scenario of a micro-manufacturing process. In this real industrial setup, the main goal is to control micro forces in a micro-drilling process.

4.1 Artificial cognitive architecture

In the following paragraphs we will try to explain all the considerations about the design and development of the artificial cognitive architecture we want to create. Firstly, the requirement analysis, differentiating between functional and non-functional requirements, will be presented. Then we will explain the most important decisions we have taken at the moment in which we made the design. Along with these decisions, some important details of the class diagram will be presented. Finally, some important details of the implementation will be discussed.

4.1.1 Requirement analysis

Hereafter we present the requirement what our artificial cognitive architecture must accomplish. We distinguish between functional requirements, which defines a function of the system, and non-functional requirements, which rather defines qualities of the system.

Functional requirements

RF1 **Control architecture:** the main function of this architecture is to control processes, the implementation of the architecture must allow the user to assign a process to the architecture and prepare the architecture to control it.

RF2 **Modules and models:** the architecture will have several modules in which there will be models to control a process, e.g., a module can contains two *single loop* models and other can contains one *direct* and one *inverse* model.

RF3 **Modes:** the architecture will run in different modes. A mode can be defined as a configuration of the different elements of the

architecture (models, reference values and process entity) to control a process. The application will be able to change between modes while running.

RF4 **Adaptation:** the application must provide a component to choose the models that a mode will need. The component choice may be due to accomplish different objectives.

RF5 **Optimization:** the architecture must provide an option to optimize the control models with a simulation model of the physical process to control. With this action, the architecture will be able to improve its behavior versus different processes.

RF6 **On-line learning:** like the optimization, the architecture must provide a mechanism that allows the user to execute a learning algorithm while the architecture will be controlling a process. Again, this mechanism will improve the behavior of our architecture.

RF7 **Objectives:** the architecture must allow that the user inserts the objectives he wants to achieve, e.g., *productivity*, *performance*, etc.

RF8 **Data types:** the architecture must allow different data types, such as integer, double or string.

Non-functional requirements

RNF1 **Middleware:** the architecture shall be quite generic and flexible to allow the user to use it over a middleware, for instance, the user may want to use the architecture to control a process in a different place, i.e., he may want to distribute our architecture to control remote process.

RNF2 **Extensibility:** the architecture shall be designed to ease the tasks of adding models, control algorithms, optimization or learning mechanism, etc.

4.1.2 Design

In order to meet with the requirements previously mentioned we have designed an object-oriented library. Along with the general classes and interfaces, some classes are provided to ease the tasks of instantiation of the architecture. In the following pages we will explain both, the main classes and interfaces and the auxiliary, but useful, classes.

The whole library is contained in the package called `cognetcon` which contains several packages with different functions, as we can see in table 1.

Name	Description
`app`	Package that contains all the classes that represent the components of the artificial cognitive architecture as well as classes that represent the application to control processes.
`data`	Package that contains the classes that represent a variable set in the architecture.
`exceptions`	Package that contains all the exception thrown in the architecture.
`model`	Package that contains an interface that represents a model in our architecture an auxiliary class to ease the implementation.
`process`	Package that contains interfaces that represents a process in our architecture and a process observer as well as some auxiliary class to ease the implementation.
`utils`	Package that contains a Log class to show information while running.

Table 1: Main packages of the designed library.

As it is expected the most important packages are `app`, `data`, `model` and `process`, so these are the packages that we will explain. We will explained them from lower to higher complexity.

Data

Name	Description
`VariableInfo`	Represents the information of a variable.
`VariableSet`	Represents a set of variables.

Table 2: Overview of the `data` package.

In this package are the classes necessary to represent a set of variables in our architecture. Firstly, `VariableInfo` represents the information of a given variable, namely, its type and its name. With this information a variable is uniquely determined, so a `VariableInfo` is perfect to identify variables. Using this class a `VariableSet` is built. This class provides different methods to create and work with sets of variables

such as sums two sets, multiply by a scalar, etc. As it is supposed, each variable inside the set is identified by its name and type, i.e., its `VariableInfo`.

With these data models we achieve the **RF8**, i.e., it is possible to use different data types to use in our architecture.

Model

Name	Description
`AbstractModel`	Auxiliary class to ease the task of instantiation.
`Model`	Represents a model of the architecture.
`ModelType`	Represents the possible types of a model.

Table 3: Overview of the `model` package.

The `Model` interface provides the sufficient methods to do almost everything we can need to do with a model. The `ModelType` enum represents the possible types of model that our architecture accepts: *single loop*, *mirroring*, also named direct models, *simulation* and *anticipation*, named inverse models.

To ease the tasks of instantiation the `AbstractModel` class is provided. This class implements the boilerplate methods of `Model` to let the user focus on implements the important methods.

Process

Name	Description
`AbstractProcess`	Auxiliary class to ease the task of instantiation.
`Process`	Represents the process in the architecture.
`ProcessLogic`	Auxiliary class used by AbstractProcess
`ProcessObserver`	Represent a process observer.

Table 4: Overview of the `process` package.

The process to be controlled by our architecture is represented by the `Process` interface. As we will see in an example, each process we want to control has to be encapsulated in a class that implements this interface. The `ProcessObserver` interface represents a process

observer, this interface has to be implemented by those classes which needs to know the state of the process as well as its outputs.

In order to ease the instantiation of a process an `AbstractProcess` class is provided. This class implements almost all the methods of `Process` and run the process's logic in a different thread. This logic is represented by the `ProcessLogic` interface.

App

The `app` package which contains two interfaces, `App` and `AppObserver`, an auxiliary class, `AbstractApp`, and two more packages, `executiveLevel` and `cognitiveLevel`, an overview of the two packages can be seen in table 5 and 6, respectively.

The `App` interface represents the control application itself, namely it will be the object that interconnects all the components of the architecture and coordinates them to control a process. It will be the access point to work with the artificial cognitive architecture. The `AppObserver` is an interface that represent an observer subscribed to the application, normally it will be extended for classes that need to know what is happened in the architecture, e.g., when a control action is sent to the process. The `AbstractApp` implements several simple methods of the `App` interface to ease the task of instantiation.

Name	Description
`AbstractExecutionManagemen`	Auxiliary class to ease the task of instantiation.
`ExecutionConfiguration`	Represents an execution configuration of the architecture.
`ExecutionManagement`	Represents the execution management of the architecture.
`Module`	Represents a module of the architecture.
`SimpleModule`	An useful implementation of module to be used directly.

Table 5: Overview of the `executiveLevel` package.

For a better understanding of the architecture, we begin explaining the `executiveLevel` (table 5). In this package we can find three main interfaces, `Module`, `ExecutionConfiguration` and `ExecutionManagement`. The `Module` represents, mainly, a container of models, so the modules are very important in our architecture. However, the power of a module relies on the models it contains, i.e., an empty module is nothing for our architecture. Each model has a type and a module is not restricted to contain only models of one type, e.g., a module can contains

two direct models and one inverse model. Due to, mainly, a `Module` is only a repository of models an implementation is provided in class `SimpleModule`. This class implements all the necessary methods and ensures that each execution has an independent copy of each model that it needs. This interface along the `Model` interface satisfy the **RF2**.

The second interface, the `ExecutionManagement`, represents the component that maintains the execution threads of the architecture. These threads are independent of the other components of the architecture and always try to run in the same frequency of the process to provide a better control. Then, the function of the `ExecutionManagement` is to interpret the `ExecutionConfiguration` and execute the models of each configuration in the right order with its correct inputs. In order to ease the implementation of a `ExecutionManagement` an abstract class, `AbstractExecutionManagement`, is provided. This class implements some simple methods of its interface in order the user can focus on the important methods.

Finally, the `ExecutionConfiguration` is an auxiliary class that the `ExecutionManagement` uses to retrieve the information of how the models are connected in a given mode.

The components where the self-capabilities will be implemented are in which we call `cognitiveLevel` (table 6). But, before begin to describe these components, we will explain the called *modes* of the artificial cognitive architecture. A mode in the architecture is simply a topography, or a pattern to interconnect some models. With this conception, a mode can be represented graphically by a graph like the common single loop of the control theory is represented (fig 6). This conception permits to interconnect different models to achieve different results in order to cover the objective. The class that represents a mode in the architecture is `Mode`. It is important to note that a mode does not specify a concrete model, it only specifies the type of models that are connected. All these considerations targets the base of the **RF3**.

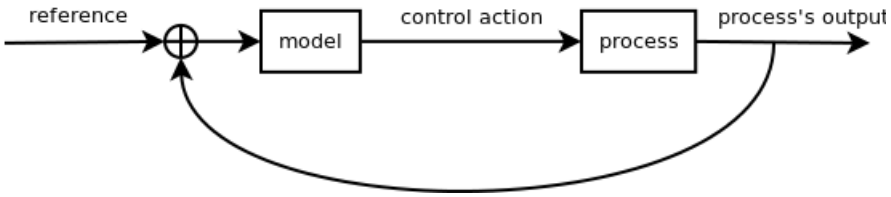

Figure 6: General single loop graph.

The `Organization` in the architecture has the function of loading the modes in the `ExecutionManagement` and switching between modes when it is necessary. To switch between modes the `Organization`

Name	Description
`AbstractAdaptation`	Auxiliary class to ease the task of instantiation.
`AbstractLearning`	Auxiliary class to ease the task of instantiation.
`AbstractOptimalSearching`	Auxiliary class to ease the task of instantiation.
`AbstractOrganization`	Auxiliary class to ease the task of instantiation.
`Adaptation`	Represents the adaptation of the architecture.
`Evaluation`	Represents the evaluation of the architecture.
`GoalManagement`	Represents the goal management of the architecture.
`Learning`	Represents the learning of the architecture.
`Mode`	Represents a mode of the architecture.
`OptimalSearching`	Represents the optimal searching of the architecture.
`Organization`	Represents the organization of the architecture.
`PerformanceIndex`	Represents a performance index that will be used by the `Evaluation`.
`TrackValue`	Represents a set of ideal values that the `GoalManagement` has to track.

Table 6: Overview of the `cognitiveLevel` package.

has to notify to the `ExecutionManagement` this change. However, the `Organization` does not known what models has to use in each mode, this task corresponds to the `Adaptation`. In this manner, the mission of the `Adaptation` is ideally choose the best models for each mode dependent of circumstances, e.g., the process. This translations in the architecture corresponds to change between `Mode` and `ExecutionConfiguration`. In addition, the `Adaptation` is prepared to run on-line to receive information in real-time and be able to execute one algorithm of self-adaptation to change the model parameters or, if it is necessary, change the model itself. With these two components along with the `Mode` class, the requirements **RF3** y **RF4** are achieved.

In order to provide the capability of running an optimal searching to find the best parameters for a given mode the `OptimalSearching`

interface is provided. It presents all the necessary methods to implements an algorithm of off-line optimization. In this manner the **RF5** is covered.

The `Learning` interface makes possible the implementation of a learning algorithm in our architecture. This interface presents the methods that are needed to implements an algorithm of on-line learning thus achieving the **RF6**.

In order to aid in the tasks that these components perform, a component that measures the behavior of the system according to different goals is presented, the `Evaluation`. This component provides methods to easily compute a performance index that indicates the behavior of the system. Thanks to this index the other components can know if they have to act or not. To provide a common interface to easily use different performance indices in our architecture, the class `PerformanceIndex` is created.

The component that translates between user goals and performance indices is the `GoalManagement`. The main role of this component is to parse the user goals and translates them, using some algorithm, into a combination of one or more performance indices. It is important to note that this is a very complex component and its correct implementation is out of our scope, due to that it is possible that in future versions of the architecture its interface may change. For now, its interface provides methods that permit a user with technical knowledge fix some objectives such as setting the *setpoint* to a fixed value. To represent this *tracking* objectives the `TrackValue` interface is created. The features of the `GoalManagement` make it possible to achieve the **RF7**.

In order to ease the instantiation of our architecture four abstract classes are provided:

- `AbstractAdaptation`.
- `AbstractLearning`.
- `AbstractOptimalSearching`.
- `AbstractOrganization`.

For a more complete insight, the whole class diagram can be found in appendix A. A simplified graphical representation of the whole architecture is represented in figure 7. With all these considerations we achieve the **RF1**, i.e., we design an architecture that is capable to control a process.

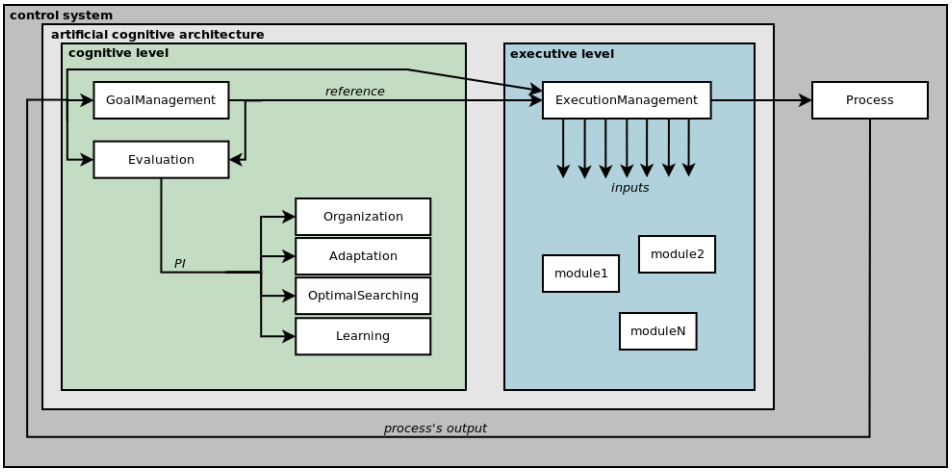

Figure 7: Graphical representation of the architecture.

4.2 Particular instantiation of the architecture

In order to show a real example and to test our architecture a particular instantiation have been developed. The instantiation is a simplified version of the COGNETCON architecture which will be distributed in order to perform a remote control. We hope that it shows the potential of the architecture as well as its flexibility and scalability.

In order to achieve this task we had to follow the general steps to instantiate our architecture particularizing when it is needed:

1. Implementation in Java of the classes that we needed. It is recommended to extends from the abstract classes presented in the architecture. The number of classes or their complex depends of the tasks we want to realize.

   In our particular case, as we will present below, we had to follow different steps, mainly they are:

   (a) Implementation of the inference models using the technology SWIG to take advantage of the models implemented in C++ by our research group.

   (b) Implementation of a self-optimization algorithm based on the CE method.

   (c) Implementation of a on-line learning algorithm based on the *Q-learning* algorithm.

2. Implementation of the Ice classes needed to make the units of our architecture distributed. Particularly, we have followed the delegation pattern to make this task easier.

3. Deployment of the *cognitive unit* in the Raspberry Pi 1.
4. Deployment of the *executive unit* in the Raspberry Pi 2.
5. Deployment of the *process unit* in the process host.
6. Adjusting and tuning models' parameters with simulation.
7. Adjusting and tuning models' parameters with experimentation and tests in a real manufacturing environment.

In order to simplify our implementation work we have taken several considerations that it is necessary to be explained. For the sake of clarity, we are grouping this considerations in different parts:

Models

In this instantiation we will use four different models. Three of them are coded in C/C++, so we have used the SWIG tool for being able to use them from Java.. The models that are exported from C/C++ are a fuzzy controller, an ANFIS direct model and an ANFIS inverse model. In this manner, the implementation of the models simply used them in a class that extends `AbstractModel`, an example of this can be seen in listing 3.

The other model is the simulation model that uses the same equation than the simulation process used to test our application (equation (22)).

Process

Two processes will be implemented for testing scenarios, one for the simulation case and other for the real case.

The implementation of the simulation process is very straightforward but for the process that represents the real machine a bit more work is needed. The connection between the machine and the process is done via Dynamic Data Exchange (DDE) and some steps are needed to configure the machine before start it. A simplified snippet of code can be found in listing 4.

Modes

We will use three modes of the MSCM (see section 3.1.1). A simplified graph-shape version of these modes is depicted in figures 8, 9 and 10 and the implementation can be found in the listings 5, 6 and 7, respectively.

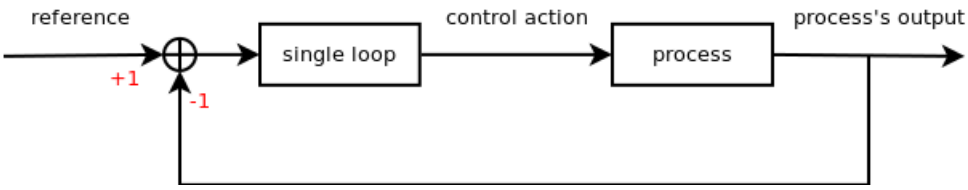

Figure 8: Graph version of Single Loop mode.

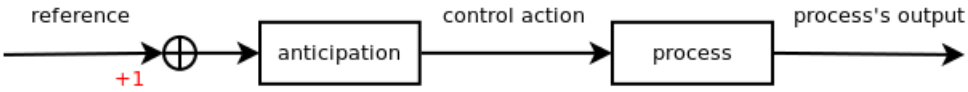

Figure 9: Graph version of the Anticipation mode.

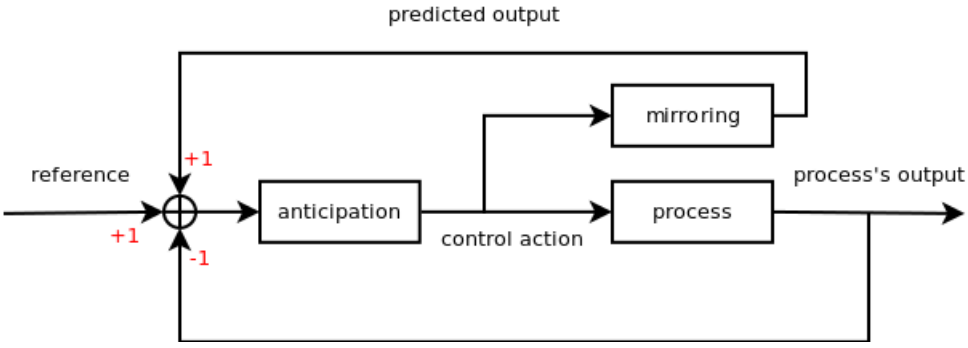

Figure 10: Graph version of the Anticipation+Mirroring mode.

GoalManagement

Goal management design is out of our TfG, only a simplified version of it is implemented. In our implementation a prefixed reference value in return in each time step.

Evaluation

In our instantiation only one performance index was considered, namely the relative Mean Squared Error (MSE). We choose this index because is a common criteria measurement used in both control theory and manufacturing processes. The equation that this performance index follows is presented in (16).

Adaptation

Due to the simplicity of our test and the fact that we will have only three models, one for each model type, the adaptation will translate between `Mode` and `ExecutionConfiguration` simply choosing the only model that is presented for each type.

It is important to note that an adaptation algorithm has not implemented for this instantiation, i.e., our adaptation is simply a translator between `Mode` and `ExecutionConfiguration`.

Organization

The organization algorithm that we have implemented is a simplified version of the algorithm of the MSCM described in section 3.1.1. The version implemented here is shown in algorithm 6.

Algorithm 6: Implemented organization algorithm.

```
1  if inTransitionZone() then
2  |  mode = SINGLE_LOOP;
3  else
4  |  if isLastActionSucessful() then
5  |  |  mode = last_mode;
6  |  else
7  |  |  simulativeOutput = computeSimulativeOutput();
8  |  |  PISL = computePI(controlSingleLoop);
9  |  |  PIAnt = computePI(controlAnticipation);
10 |  |  PIAntMirr = computePI(controlAnticipationMirroring);
11 |  |  mode = bestMode(PISL, PIAnt, PIAntMirr);
12 |  end
13 end
```

OptimalSearching

The off-line optimization algorithm that we have implemented for this instantiation is the *cross-entropy method*, presented in section 3.1.2.

Learning

In order to provide the instantiation with a on-line learning algorithm we have implemented the *Q-learning* algorithm described in section 3.1.3.

ExecutionManagement

We have done an implementation of `ExecutionManagement` that maintains each execution configuration running in a different thread. For each configuration the management gathers the inputs of each model, executes it and, if the configuration is the main configuration, the control action is sent to the process directly.

Application

Finally, the implementation of the `Application` is simply an organizer that receives the process's output and distributes it with the reference to the adaptation, organization and learning components. An interesting detail to be noted is that the process's output and the reference is distributed between the components if and only if the process's output exceeds a threshold, given by one process's parameter.

It is important to note that this instantiation has used, when it is possible, the abstract classes that are provided with the library and has coded in order to be able to run correctly in both non-distributed and distributed environments. So, in order to ease the implementation in the distributed environment we have used the *ZeroC - Ice* technology.

To use this technology, as we have explained in section 3.1.4, we have written the specification of the three units we want to deploy remotely and generate the *Ice-classes*. Then, we have to implement some classes to provide the distributed functionality. In order to do this in the easiest way, we have implemented some special classes that used the delegation pattern to use the implemented classes in a distributed way. In this manner, if we want to distribute other process or other application, the only thing we have to change is the delegate object.

In order to use *IceGrid* it is necessary to create some configuration files in order to set up the main the connection with the *registry* machine and to set up some needed parameters. An example of the file used by the registry server can be seen in listing 1 and a file used by a server application can be seen in listing 2. As we can see we have to set the address of the IceGrid registry, i.e. the `Ice.Default.Locator` value and the type of connection, the name of the node and the directory where data will be saved.

```
IceGrid.Registry.Client.Endpoints = tcp -p 4061
IceGrid.Registry.Server.Endpoints = tcp
IceGrid.Registry.Internal.Endpoints = tcp
IceGrid.Registry.Data = ice_registry/registryDB
```

Listing 1: IceGrid Registry configuration file.

```
Ice.Default.Locator=IceGrid/Locator:tcp -h <hostIP> -p 4061
IceGrid.Node.Endpoints=tcp
IceGrid.Node.Name=AppNode
IceGrid.Node.Data=ice_registry/nodes/AppNode
```

Listing 2: IceGrid AppNode configuration file.

4.3 Use-case scenarios

Once we have implemented our architecture and some classes, we are ready to test if it works as we expect. To test our application two scenarios are presented: the first is a simulation scenario to assess the connectivity and the work-flow. In this scenario we simulate a real process with a third-degree recursive function; the second is a real test in unique facilities in Madrid, we are going to control forces in micro-drilling process in order to minimize the tool wear and to maximize the metal removal rate.

For the two scenarios the same instantiation of our architecture will be used. This instantiation will be the presented in section 4.2. If some changes are needed for one particular scenario, they will be explained in the correspondent one.

The structure of the whole environment is shown in figure 28. For more information about the features of each machine or deeper explanation please see the appendix C.

As with the instantiation, if some changes are made in the structure for one scenario they will be explained in each scenario.

4.3.1 Simulation framework

In this scenario we are going to use an implementation of a simulation process to execute in the *process machine*. To simulate a process we are going to use a third-degree recursive function (22).

$$force_t = \langle \vec{action}_t, \vec{a} \rangle - \langle \vec{force}_t, \vec{b} \rangle \tag{22}$$

where $\vec{action}_t = (\hat{action}_t, \hat{action}_{t-1}, \hat{action}_{t-2}, \hat{action}_{t-3})$, $\hat{action}_t$ is the control action in time t, $\vec{a} = (\hat{a}_0, \hat{a}_1, \hat{a}_2, \hat{a}_3)$ is an adjust coefficient vector, $force_t = (f_{t-1}, f_{t-2}, f_{t-3})$, f_t is the output force in time t, $\vec{b} = (\hat{b}_0, \hat{b}_1, \hat{b}_2)$ is another adjust coefficient vector and $\langle \cdot, \cdot \rangle$ denotes the scalar product of two vectors.

The values of vectors $\vec{d}$ and $\vec{c}$ was calculated empirically and they are:

$$\vec{a} = (0.004322, 0.02467, 0.008623, 0.00002178)$$

$$\vec{b} = (-2.447, 1.993, -0.5406)$$

In order to scale up the forces during the simulation a scaling factor of 0.02 is applied to the above model. With all these considerations the equation (22) simulate a $1mm$ diameter tool in a microdrilling process with sample time of $50ms$, feed rate of $100mm/min$ and spindle spin of $28500rpm$. Approximately, the time of the experiment is between $4500ms$ and $6000ms$, for our simulation test we have chosen $6000ms$.

The main goal of this scenario is to check that the instantiation of our artificial cognitive architecture works appropriately before running it in a real environment. So, all the distributed components are deployed to test the connectivity and the data work-flow, i.e., the *Process host* of figure 28 is not connected with the *KERN Evo* machine. We have taken special care checking that the control action (*override*) stay between the allowed values accepted by the micro-drilling process to ensure a correct operation in the real scenario.

In this scenario we are going to carry out several tests in order to check that all the components of the architecture work perfectly. Thus, firstly we will execute with the three modes in active; secondly, we will execute the optimization of the *Single Loop* mode and in the third place, we will execute with the *Single Loop* mode and the learning activated, all to check if the mode switching algorithm, the optimization and the learning work properly.

4.3.2 Real time setup in an industrial environment

In this test we will use an experimental platform that includes a cutting force sensor on three axes, two vibration sensors for y, z axes and a laser sensor for measuring the variation in tool length and radius.

The measurement of cutting force signals was done with a multi-component dynamometer. A Kistler sensor (MiniDyn 9256C1) was used, with a sensitivity of below 26 pC/N on the z-axis, with a band width of up to 5 kHz. The dynamometric platform was securely clamped to the machine-tool worktable and connected to a Kistler 5070A 02100 multi-channel charge amplifier.

The vibration signals were measured by two accelerometers attached with wax to the y, z-axes to the workpiece. The sensor model on the y-axis was a 352B PCB Piezotronics with a sensitivity of 1015 mV/g and a 10 kHz bandwidth and a Deltatron 4519-003 Brüel & Kjaer sensor with a sensitivity of 10.58 mV/g and bandwidth of 20 kHz was positioned on the z-axis. Both sensors were connected to a Brüel & Kjaer 2694 series load amplifier. Also the tool length was measured on-line with a laser sensor. This high precision visible red-light laser is a state-of-the-art measuring system that is fully

up to date and commercially available. The sensor model is BLUM Laser Control NT P87.0634, designed for tool sizes greater than 5 μm. All the sensor signals were fed into a NI 6251 National Instruments data-acquisition card, with an acquisition rate of 50 kHz, and were processed by a National Instruments high-performance PXI-8187 embedded controller. Furthermore, the position of the tool tip (x, y, z) was obtained via the ethernet connection of the CNC of the machine, using a sampling frequency every 12 ms (83.33 Hz).

All cutting operations were done in a Kern Evo Ultra-Precision Machine Centre (see Table I), equipped with a Heidenhain iTNC540 CNC. Maximum spindle speed (n) and feedrate (f) were 50,000 rpm and 16,000 mm/min respectively.

In order to give some graphical description of the described platform the figure 11 is presented.

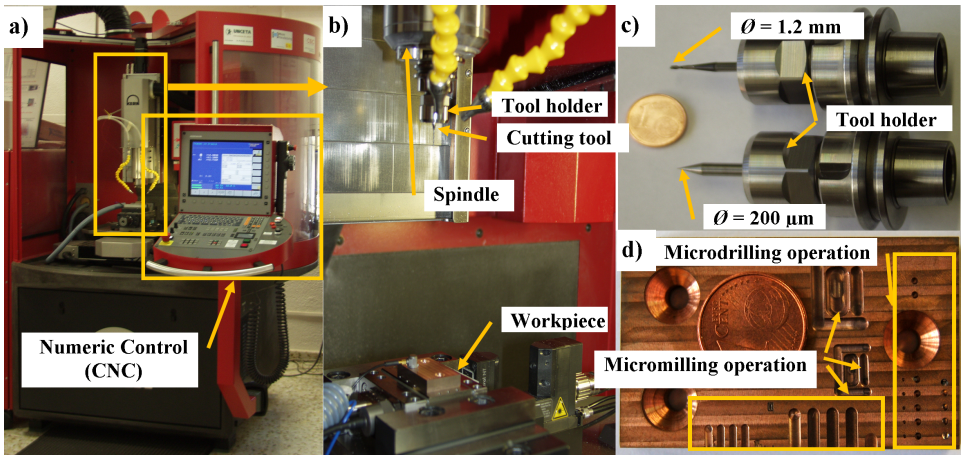

Figure 11: KERN Evo parts description.

In this scenario we are going to perform a simple control of the process's force. This step is very important in order to demonstrate the viability of the developed architecture in a real environment.

5 Test and results

In this section we present the results obtained after the execution of the test. All the results will be presented with a graphical chart in order for a better understanding of the results.

5.1 Simulation studies

As we have said in section 4 in this environment we have checked all the components of the architecture. Firstly, in order to get a reference we are going to monitor the simulation process without control it. The results of the monitoring are shown in figure 12. As we can see, the mean of the value is around $14N$. We are trying to down this force to $10N$ using our instantiation of the architecture.

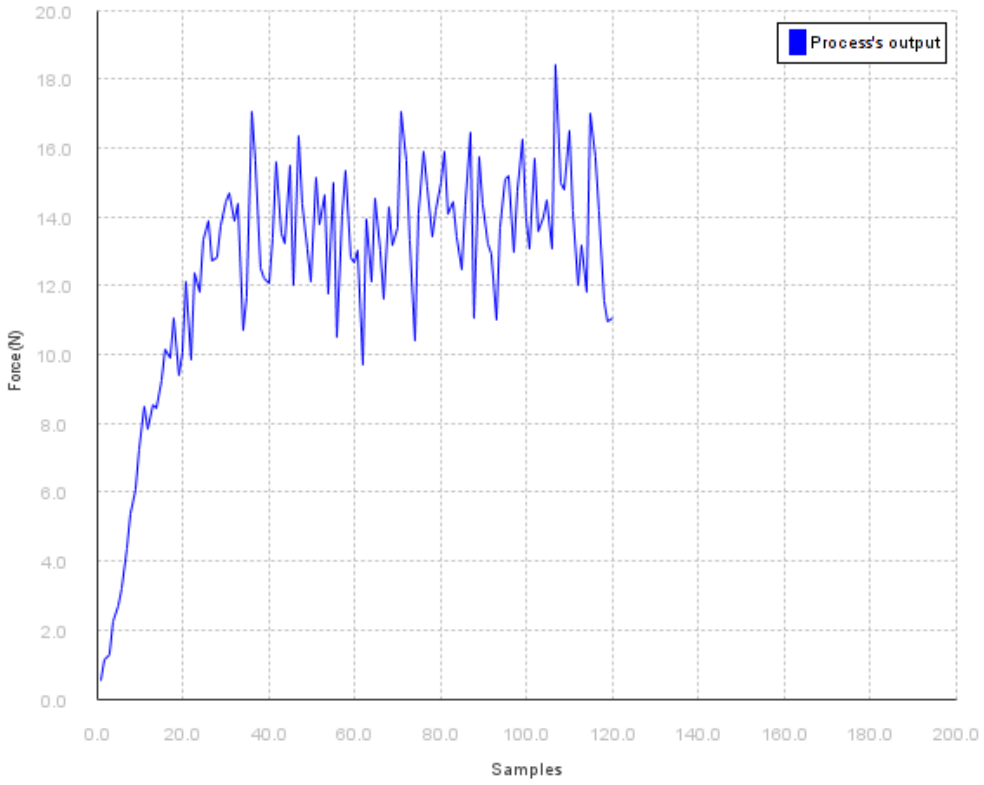

Figure 12: Monitoring of the drilling force in a simulation study.

The results obtained executing the control over five continued experiments with the three modes enabled are presented in figure 13. We can see how the mode are changing in the chart and how the process's output drastically falls to stay around 10 $11N$. These are really good results that

demonstrate that the mode switching algorithm as well as the control itself is running correctly.

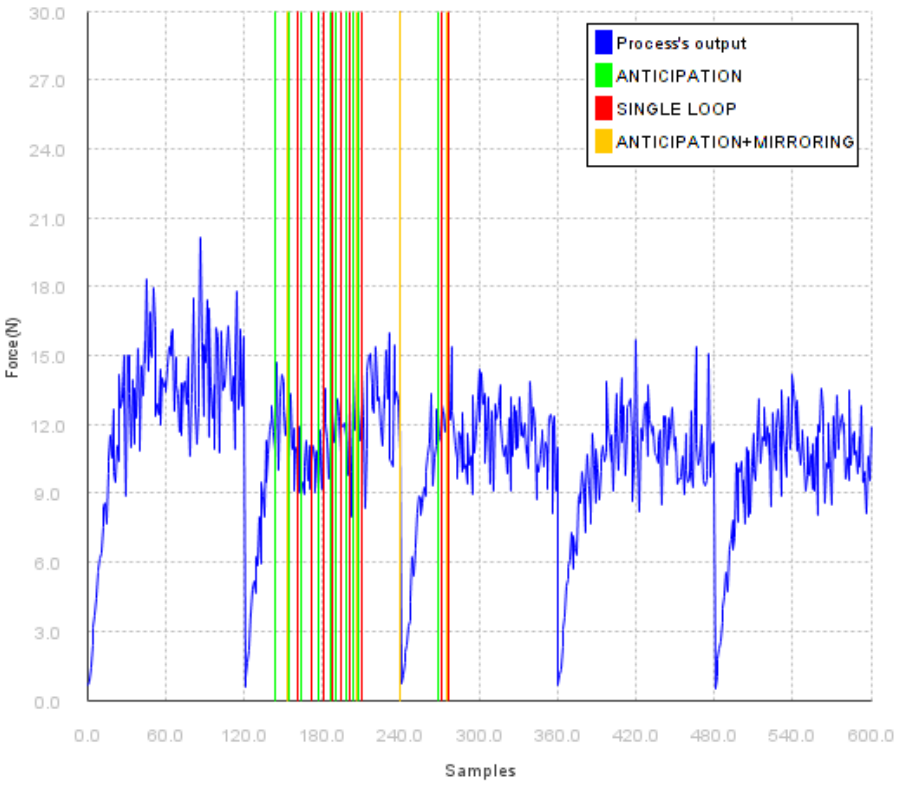

Figure 13: Behavior of the drilling force in a simulation study when the cognitive control is enabled.

Once we have checked that the mode switching and the control are correct we are going to test one of the most important component: the optimal searching. The optimal searching execute the CE method with 100 iterations changing the parameters of the fuzzy controller used in the single loop model. The parameters of the fuzzy controller, before the optimization, were $KE = 1.135$ and $KDE = 0.3159$. After 100 execution of the CE method the parameters that achieve a better performance index were $KE = 6.60700$ and $KDE = 0.09703$. In order to check the validity of this result we do another experiment with the obtained parameters. The results of this experiment are shown in figure 14. In this experiment the first execution is without control and the four consecutive executions are performed only with the single loop mode. With these results we can say that the parameters obtained are valid then the optimal searching component is correct.

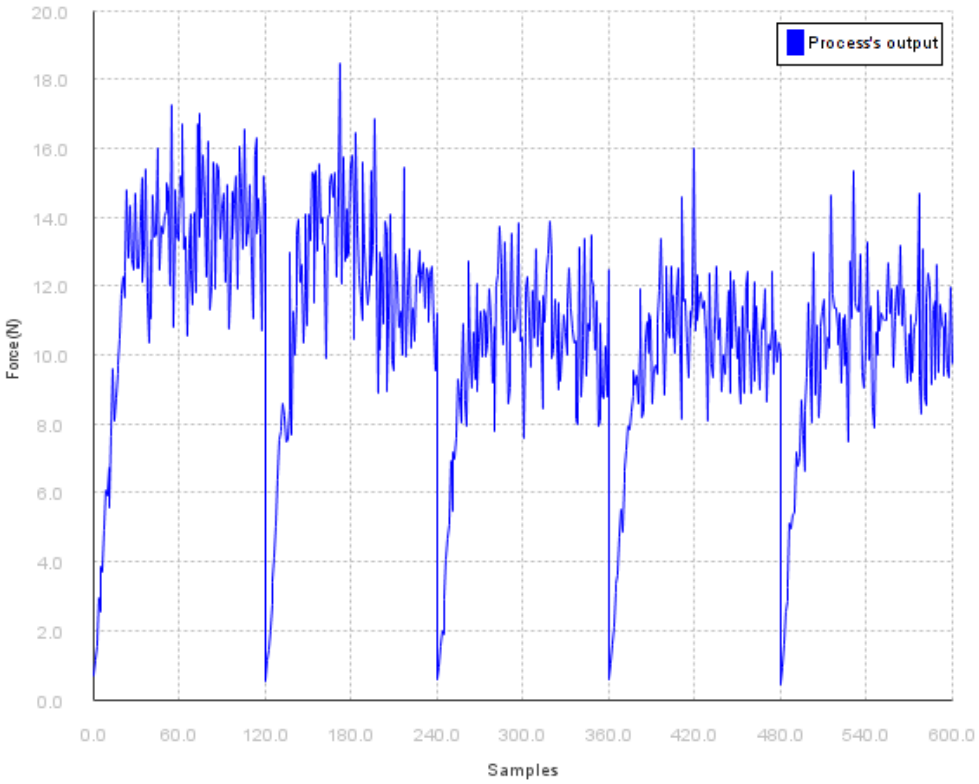

Figure 14: Behavior of the drilling force in a simulation study after optimization of parameters.

Finally, we tested the learning capability of the architecture. The results of this experiment are shown in figure 15. The obtained results are very similar to those obtained with the optimization, but for the propose of this experiment, the learning produce a good control that improves over the time.

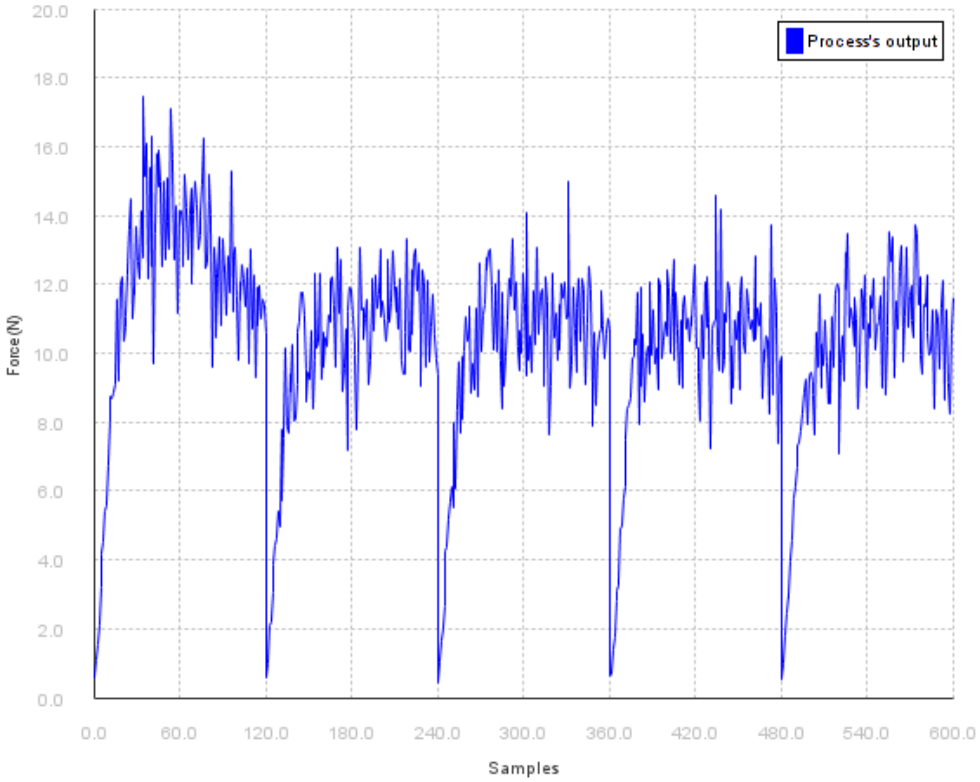

Figure 15: Behavior of the drilling force in a simulation study when the reinforcement learning is activated.

5.2 Real time test in a manufacturing plant

The goal of this scenario is demonstrate that a low-cost hardware architecture can serve to remote control a real micro-drilling process. In order to prove that we have done several experiments, here we present the results of two of them. The first experiment corresponds with one of the first experiments that we realized. The process's output without control action can be seen in figure 16.

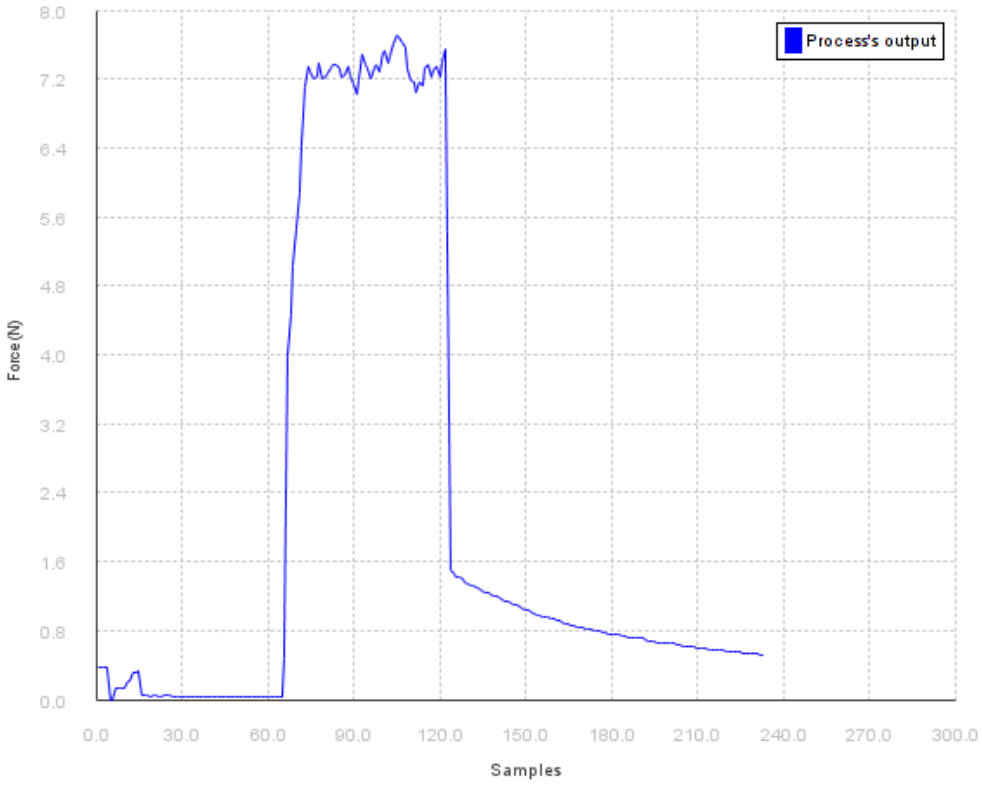

Figure 16: Monitoring function running in the platform (first run). Behavior of real time micro-drilling process.

As we can see the output is around $7N$, so we set the reference value in $6N$. The results obtained in this experiment are shown in figure 17.

Although the control is only showed in the final section, we can see that the architecture is really controlling the process's output. This delay in the control is due to the delays in the communication and the low computational power of the Raspberry Pi.

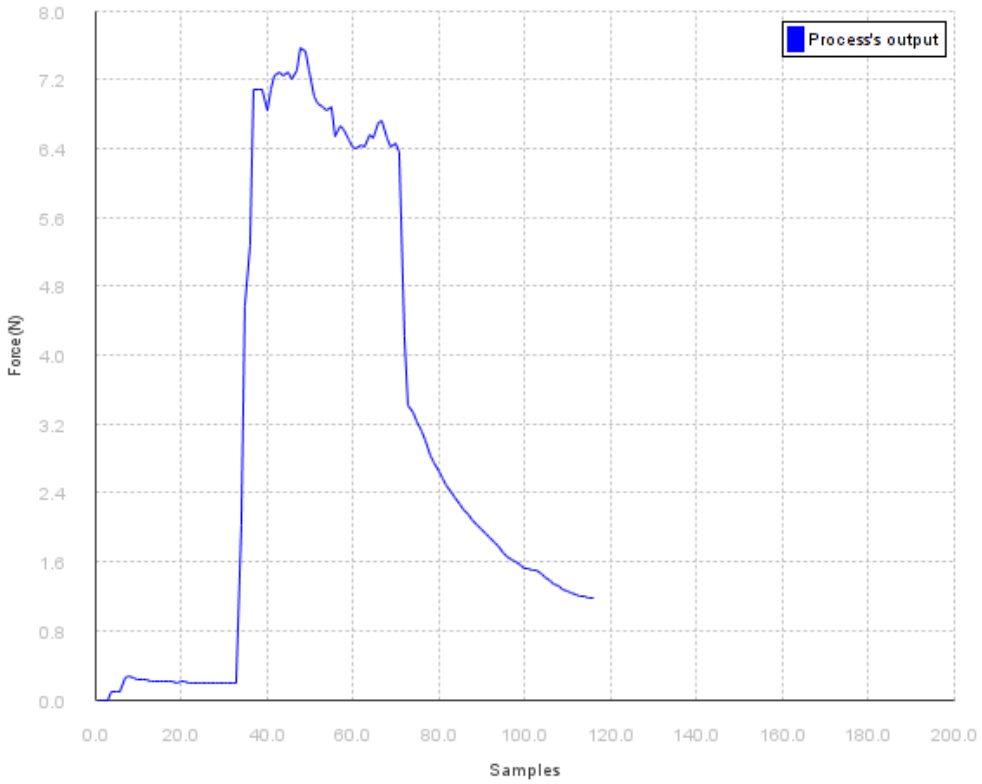

Figure 17: First run of cognitive control of drilling force in micro-drilling process ($setpoint = 7N$).

The second experiment that we present is one realized when the tools has done several drills, so it presents some wear. Due to that the force is higher than the previous experiment. As with the first experiment two results are presented. The first is the process's output without action control (figure 18) and the second is the process's output when the architecture is controlling (figure 19). As we can appreciate the results without action control present values around $10N$, so we set a reference value of $8N$ to test the control. Comparing the two results we can say that these results are even better than the first ones.

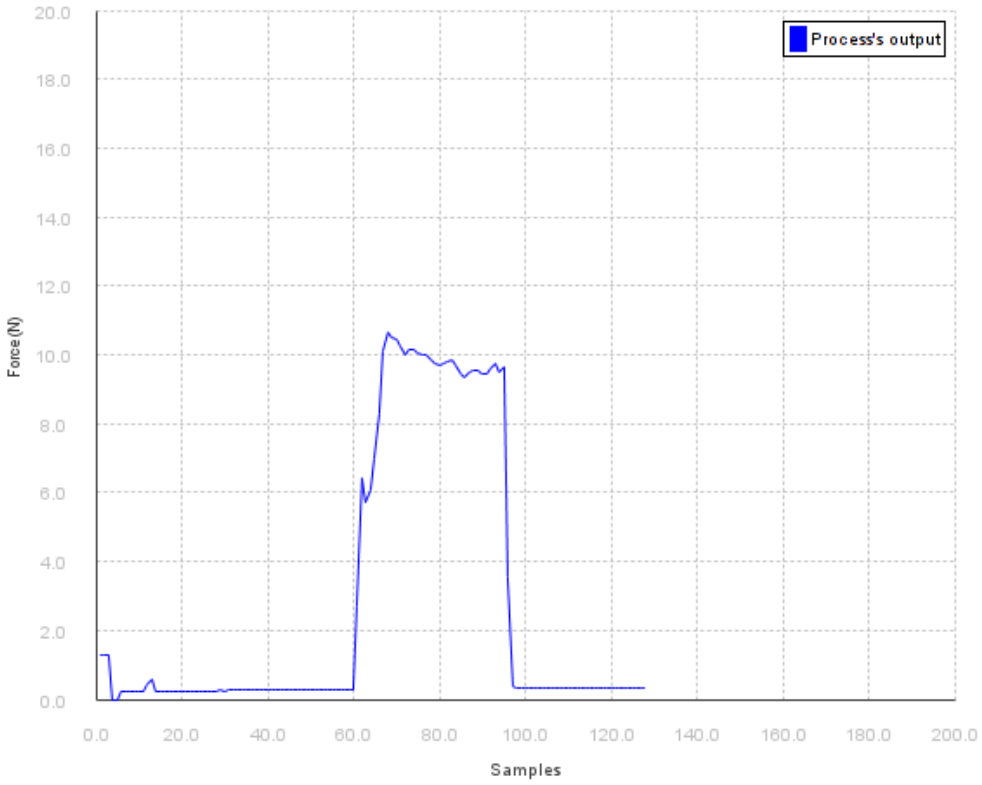

Figure 18: Monitoring function running in the platform (second run). Behavior of real time micro-drilling process.

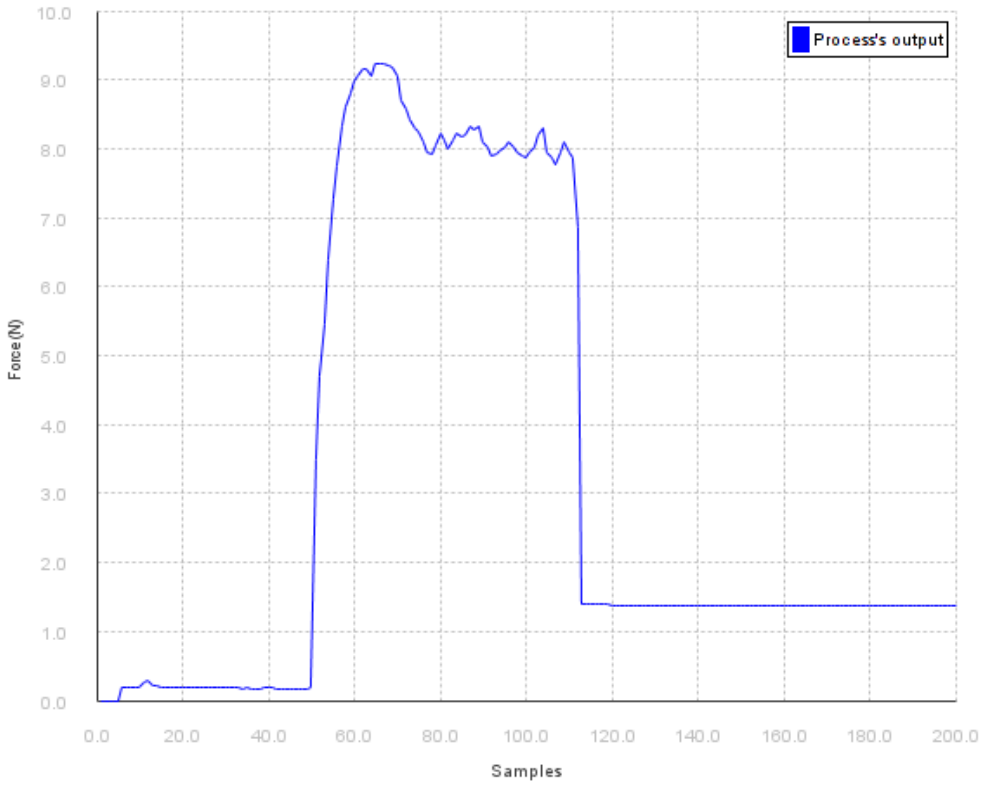

Figure 19: Second run of cognitive control for regulating force in a micro-drilling process ($setpoint = 8N$).

6 Conclusions and future work

In order to expose the conclusions of our work we are going to summarize all the important tasks that we have done. Firstly, an instantiation of the artificial cognitive architecture is designed and developed. In order to ease this task we have used inference models developed by our research group with the help of SWIG. To provide our instantiation with the ability of self-optimization and self-learning we have designed and developed an off-line optimization algorithm based on CE algorithm and an on-line learning algorithm based on *Q-learning* algorithm.

In addition, we developed classes that provide this instantiation with the ability of running in a distributed manner. In order to ease the task of environment configuration we have use the *IceGrid* service to avoid the task of entering IP directions of the hosts when we use the instantiation.

Finally, we test the instantiation in a simulation environment and in a real manufacturing environment, obtaining in both very good results. It is important to note that in the experiments we are using low-cost platform hardware, Raspberry Pi, to run our architecture. This gives our work an important added value.

So, relying on the presented results we can say that an artificial cognitive architecture build on low-cost platform hardware with viability to implement control system has been developed. With all of this the functional requirements are totally achieved as well as the non-functional requirements, due to the ease of building the instantiation (**RNF1**) and building it using a middleware (**RNF2**).

All the realized work gives us an important set point from which we can do a lot of future work, centering in the implementation of control system built on low-cost hardware.

We are aware of the fact that much work remains to be done in order to achieve the ideal artificial cognitive architecture, so the next objectives in the near future are:

- To design and develop a practical goal management complement.
- To add more models to our repository to be able to do more complex tests.
- To implement an easy way to do a deployment of the architecture without an user intervention.
- To improve the way in which our instantiation execute the components to achieve a better performance using the Raspberry Pi.
- To realize more complex test in order to prove the viability of the developed architecture in more environment.

7 References

[1] A. Meystel. Architectures for intelligent control systems: The science of autonomous intelligence. In *Intelligent Control, 1993., Proceedings of the 1993 IEEE International Symposium on*, pages 42–48, Aug 1993. doi: 10.1109/ISIC.1993.397726.

[2] D. Vernon, G. Metta, and G. Sandini. A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *Evolutionary Computation, IEEE Transactions on*, 11(2):151–180, April 2007. ISSN 1089-778X. doi: 10.1109/TEVC.2006.890274.

[3] Susan Hurley. The shared circuits model (scm): How control, mirroring, and simulation can enable imitation, deliberation, and mindreading. *Behavioral and Brain Sciences*, 31(01):1–22, 2008.

[4] Alfonso Sánchez Boza, Rodolfo Haber Guerra, and Agustín Gajate. Artificial cognitive control system based on the shared circuits model of sociocognitive capacities. a first approach. *Engineering Applications of Artificial Intelligence*, 24(2):209–219, 2011.

[5] Yujia Wang and Yupu Yang. Particle swarm optimization with preference order ranking for multi-objective optimization. *Information Sciences*, 179(12):1944–1959, 2009.

[6] Jinhua Zhang, Jian Zhuang, Haifeng Du, and Sun'an Wang. Self-organizing genetic algorithm based tuning of pid controllers. *Information Sciences*, 179(7):1007–1018, 2009.

[7] Reuven Y Rubinstein and Dirk P Kroese. *The cross-entropy method: a unified approach to combinatorial optimization, Monte-Carlo simulation and machine learning*. Springer, 2004.

[8] Andre Costa, Owen Dafydd Jones, and Dirk Kroese. Convergence properties of the cross-entropy method for discrete optimization. *Operations Research Letters*, 35(5):573–580, 2007.

[9] Dirk P Kroese, Reuven Y Rubinstein, and Thomas Taimre. Application of the cross-entropy method to clustering and vector quantization. *Journal of Global Optimization*, 37(1):137–157, 2007.

[10] Hamid Boubertakh, Mohamed Tadjine, Pierre-Yves Glorennec, and Salim Labiod. Tuning fuzzy pd and pi controllers using reinforcement learning. *ISA transactions*, 49(4):543–551, 2010.

[11] Kai-Tai Song and Te-Shan Chu. Reinforcement learning and its application to force control of an industrial robot. *Control Engineering Practice*, 6(1):37–44, 1998.

[12] Evangelos Theodorou, Jonas Buchli, and Stefan Schaal. A generalized path integral control approach to reinforcement learning. *The Journal of Machine Learning Research*, 9999:3137–3181, 2010.

[13] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.

[14] EP James, MJ Tudor, SP Beeby, NR Harris, P Glynne-Jones, JN Ross, and NM White. An investigation of self-powered systems for condition monitoring applications. *Sensors and Actuators A: Physical*, 110(1): 171–176, 2004.

[15] A Flammini, P Ferrari, E Sisinni, D Marioli, and A Taroni. Sensor interfaces: from field-bus to ethernet and internet. *Sensors and Actuators A: Physical*, 101(1):194–202, 2002.

[16] Douglas C Schmidt, David L Levine, and Sumedh Mungee. The design of the tao real-time object request broker. *Computer Communications*, 21(4):294–324, 1998.

[17] Michi Henning. Choosing middleware: Why performance and scalability do (and do not) matter, 2009. URL `http://www.zeroc.com/articles/IcePerformanceWhitePaper.pdf`.

[18] ZeroC - Ice vs. CORBA. URL `http://www.zeroc.com/iceVsCorba.html`.

[19] RY Rubinstein. A stochastic minimum cross-entropy method for combinatorial optimization and rare-event estimation*. *Methodology and Computing in Applied Probability*, 7(1):5–50, 2005.

[20] CNET: Create a retro game console with the Raspberry Pi. URL `http://www.cnet.com/how-to/create-a-retro-game-console-with-the-raspberry-pi`.

[21] Meet Jasper: open-source voice computing. URL `http://www.raspberrypi.org/meet-jasper-open-source-voice-computing`.

[22] Unified Modeling Language. URL `http://www.uml.org`.

[23] Real time specification for java. URL `http://www.rtsj.org`.

[24] Simplified wrapper and interface generator. URL `http://www.swig.org/index.php`.

[25] Raspberry Pi model B specifications. URL http://docs-europe.electrocomponents.com/webdocs/127d/0900766b8127da4b.pdf.

[26] KERN Evo Ultra Precision CNC Machining Centre. URL http://www.kern-microtechnic.com/upload/media/kern_evo_e.pdf.

Appendix

A Class diagrams

In the following pages main class diagrams of the developed architecture are presented. For the sake of clarity in the packages `cognitiveLevel` and `executiveLevel` the abstract classes are not presented and are independently shown.

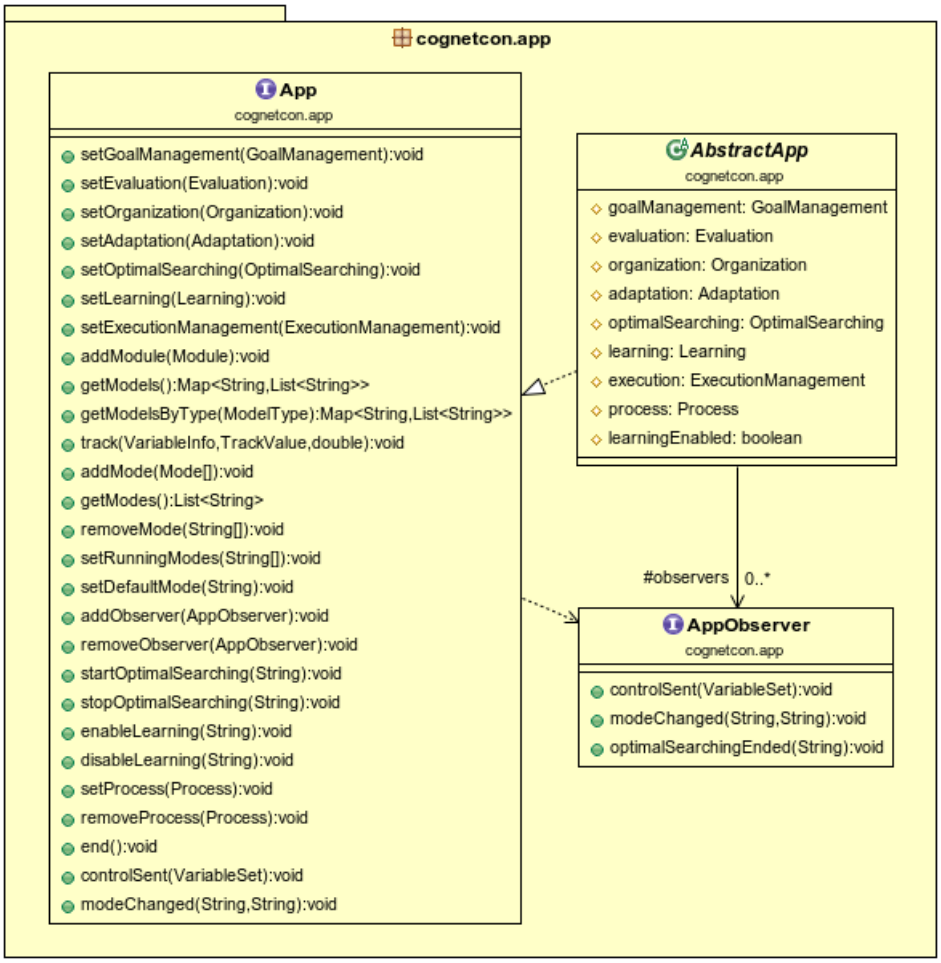

Figure 20: Class diagram of `app package`.

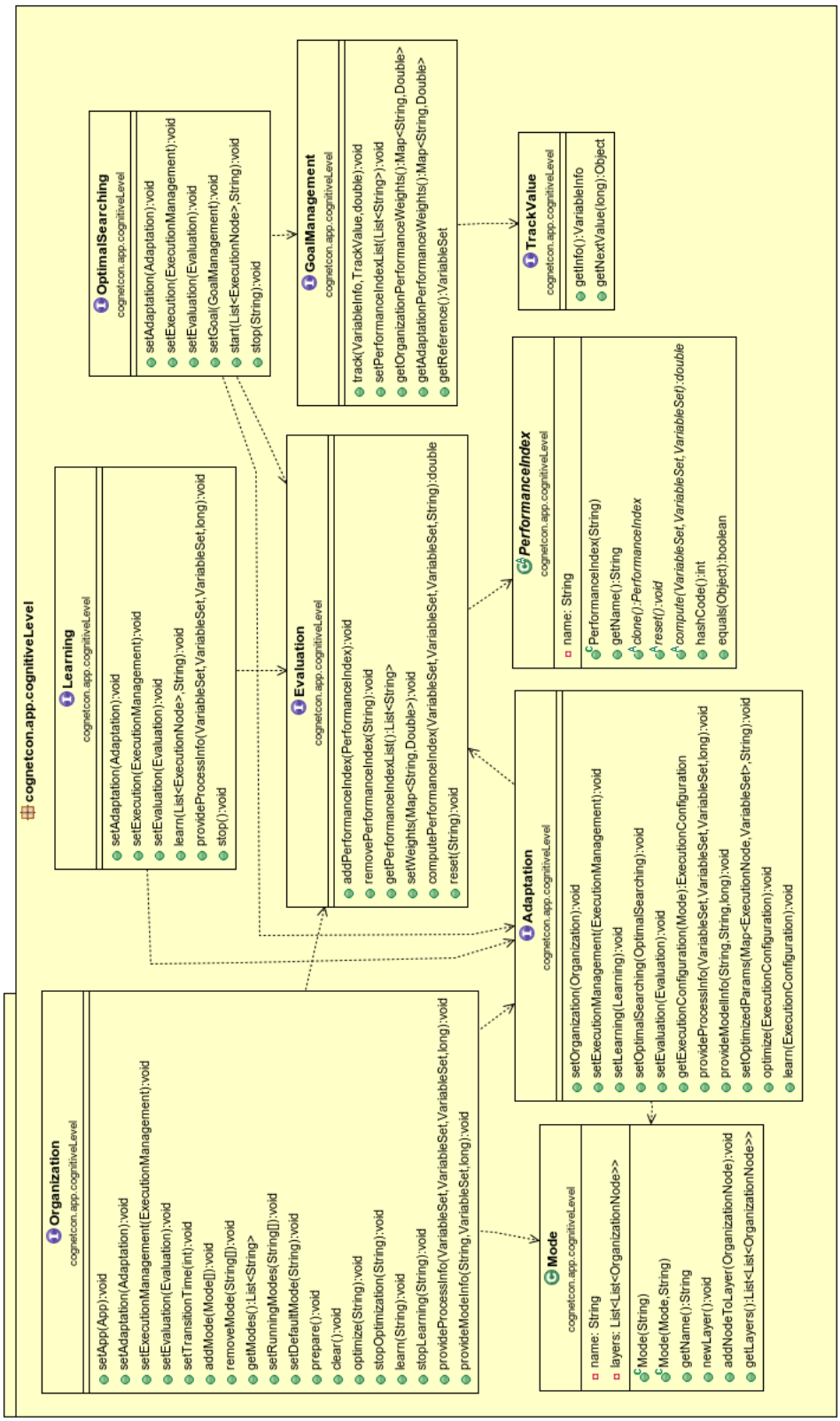

Figure 21: Class diagram of `cognitiveLevel package`.

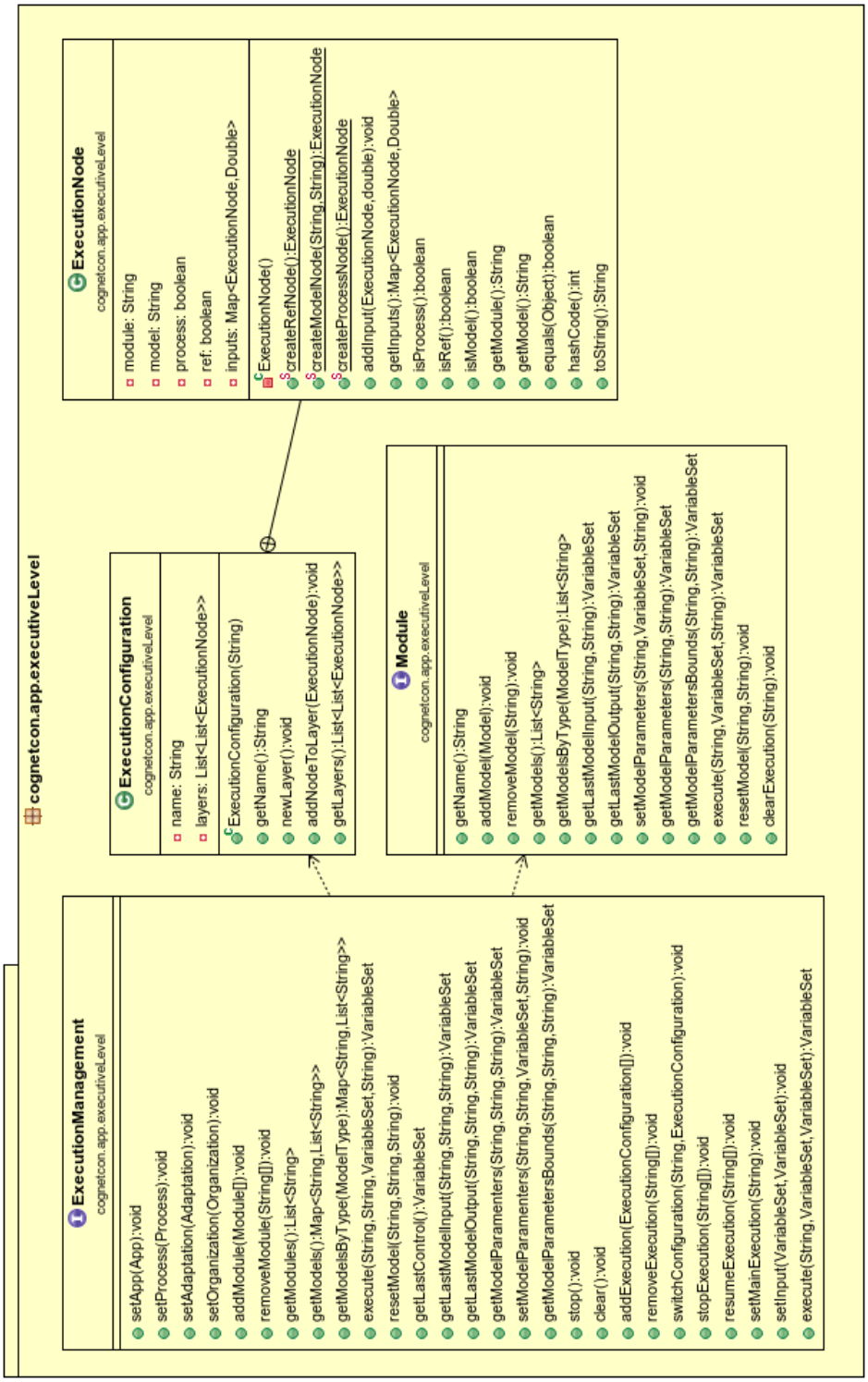

Figure 22: Class diagram of `executiveLevel package`.

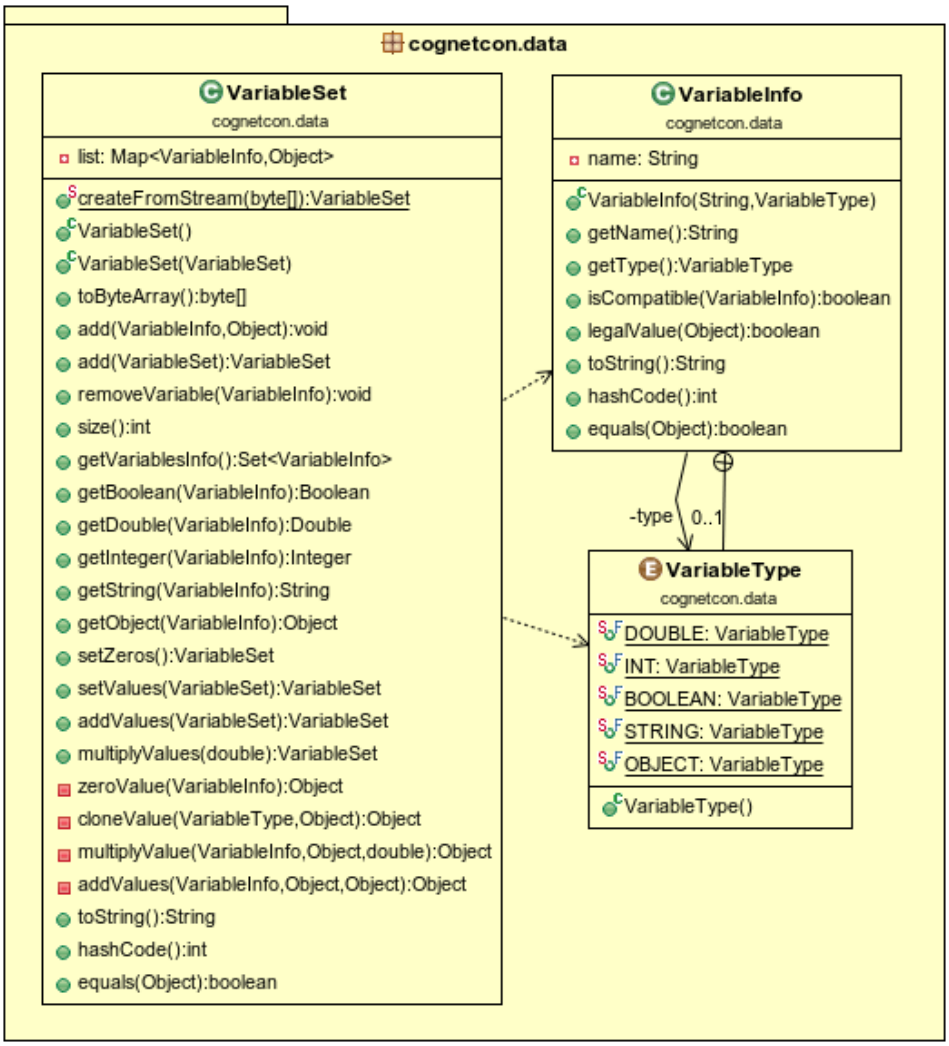

Figure 23: Class diagram of `data package`.

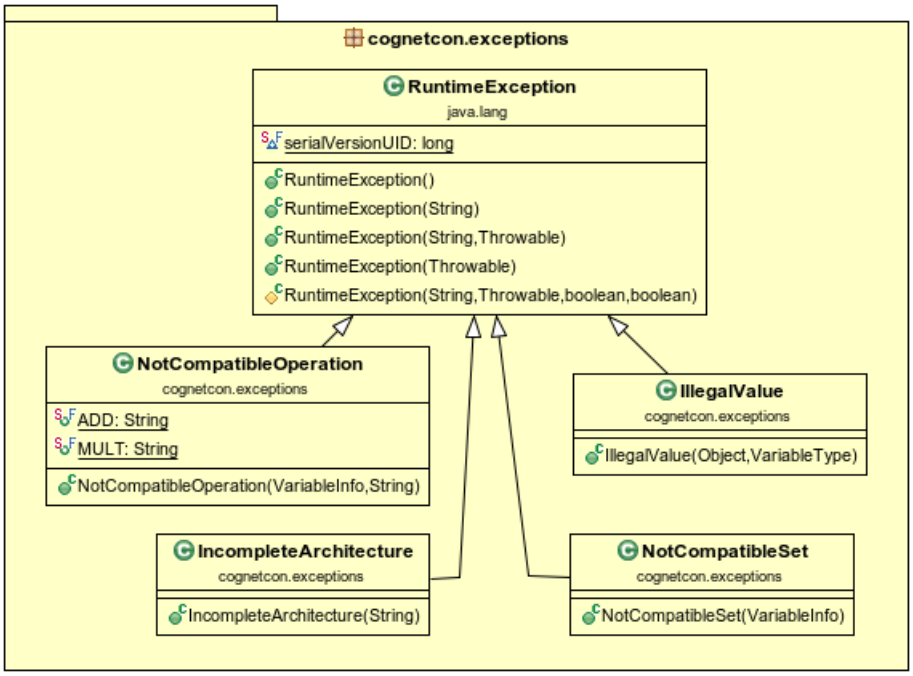

Figure 24: Class diagram of `exceptions package`.

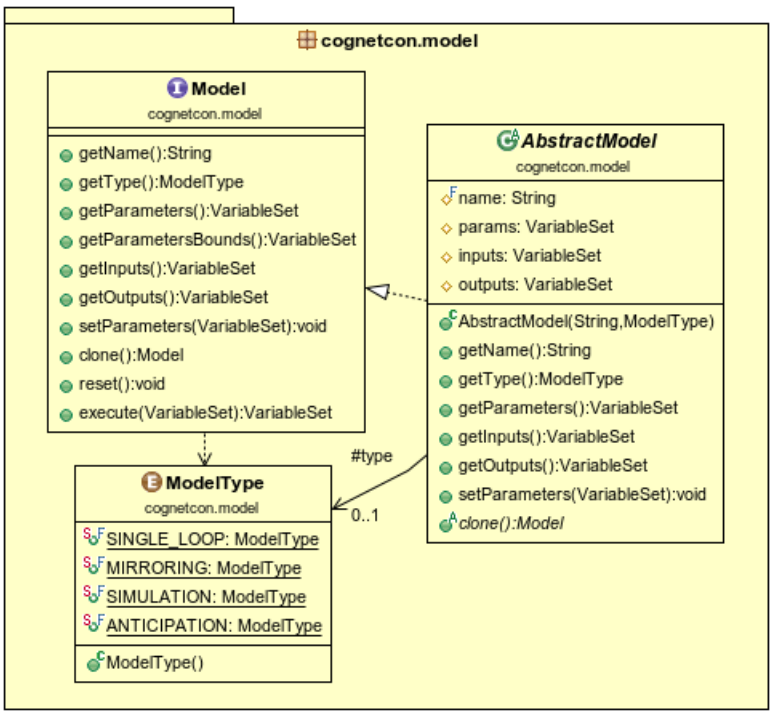

Figure 25: Class diagram of `model package`.

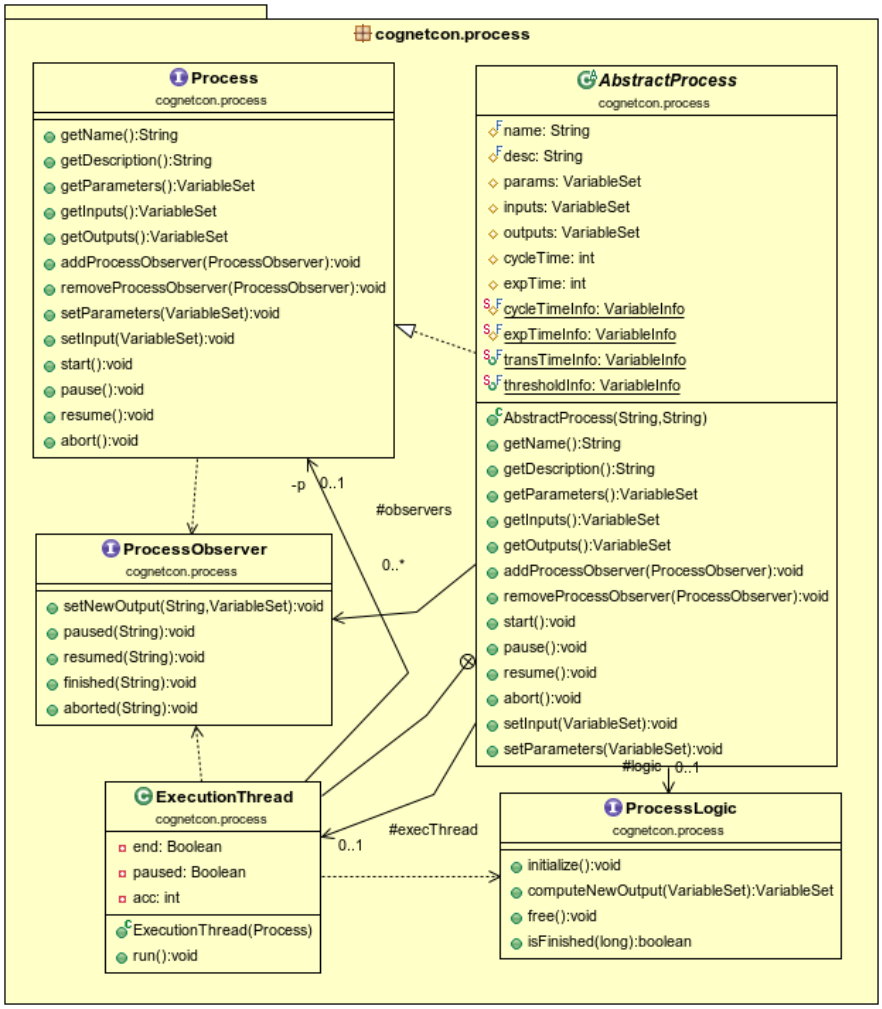

Figure 26: Class diagram of `process package`.

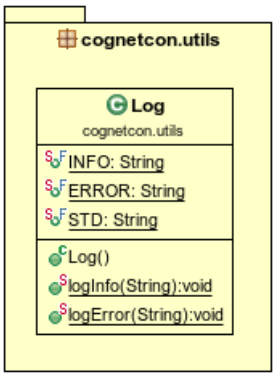

Figure 27: Class diagram of `utils package`.

B Code listings

In this appendix the referenced code listings are shown. The `try {} catch {}` statements are omitted in order to simplified the showed code.

```
1  public class FuzzyController extends AbstractModel {
2      /* Attributes */
3
4      static {
5          System.loadLibrary("C_Lib");
6      }
7
8      public FuzzyController(String confFile) {
9          super(NAME + "(" + confFile + ")", ModelType.SINGLE_LOOP);
10
11         fc = new FuzzyControl();
12         C_Lib.readFuzzyParams(fc, confFile);
13
14         params.add(KE, fc.getKe());
15         params.add(KDE, fc.getKde());
16
17         iValues = new VariableSet(params);
18         iFile = confFile;
19     }
20
21     @Override
22     public VariableSet getParametersBounds() {
23         VariableSet aux = new VariableSet();
24         aux.add(new VariableInfo(KE.getName()+"_min", KE.getType(), 0.0);
25         aux.add(new VariableInfo(KE.getName()+"_max", KE.getType(), 10.0);
26         aux.add(new VariableInfo(KDE.getName()+"_min", KDE.getType(), 0.0);
27         aux.add(new VariableInfo(KDE.getName()+"_max", KDE.getType(), 10.0);
28
29         return aux;
30     }
31     @Override
32     public void setParameters(VariableSet params) throws NotCompatibleSet {
33         super.setParameters(params);
34         setControllerParams(params);
35     }
36     @Override
37     public void reset() {
38         params.setValues(iValues);
39         setControllerParams(iValues);
40
41         fc.getIntegrator().setLastInput(0);
42         fc.getIntegrator().setLastOutput(0);
43         fc.getDerivative().setLastInput(0);
44     }
45     @Override
46     public FuzzyController clone() {
47         return new FuzzyController(iFile);
48     }
49     @Override
50     public VariableSet execute(VariableSet input) {
51         VariableSet out = new VariableSet();
52         double inputDouble = 0.0;
53         VariableInfo outInfo = null;
54         SWIGTYPE_p_double outData = C_Lib.new_double_array(1);
55
56         /* Obtaining inputDouble and outInfo */
57
58         C_Lib.fuzzyCalculate(fc, inputDouble, outData);
59         out.add(outInfo, C_Lib.double_array_getitem(outData, 0));
60
61         return out;
62     }
63
64     // Auxiliary functions
65     private void setControllerParams(VariableSet params) {
66         fc.setKe(params.getDouble(KE));
67         fc.setKde(params.getDouble(KDE));
68     }
69 }
70
```

Listing 3: Fuzzy controller implementation.

```
public class MicroProcess extends AbstractProcess {
    /* Attributes */

    public MicroProcess() {
        super(NAME, DESC);
        this.logic = new DDEMicroLogic();

        conversation = new DDEClientConversation();
        conversation.connect(SERVICE, TOPIC);

        // Inputs
        inputInfo = new VariableInfo("override (%)", VariableType.DOUBLE);
        inputs.add(inputInfo, initControl);
        // Outputs
        outputInfo = new VariableInfo("mean force (N)", VariableType.DOUBLE);
        outputs.add(outputInfo, 0.0);
        // Parameters
        params.add(s0, 0);
        params.add(f0, 0);
        params.add(lc0, 0);
        params.add(cycleTimeInfo, 100);
        params.add(expTimeInfo, 12000);
        params.add(transTimeInfo, 3);
        params.add(ref, 10.0);
    }

    @Override
    public void setParameters(VariableSet params) throws NotCompatibleSet {
        super.setParameters(params);
        /* Getting parameters */

        /* Sending parameters to the machine */
        conversation.poke("f0", f0v);
        conversation.poke("s0", s0v);
        conversation.poke("lc0", lc0v);
        conversation.poke("ciclo", ciclo);
        conversation.poke("exp_time", expTime);
    }

    @Override
    public void setInput(VariableSet control) {
        super.setInput(control);
        conversation.poke("override", Math.round(control.getDouble(inputInfo)));
    }

    private class DDEMicroLogic implements ProcessLogic {
        private VariableSet output;

        public DDEMicroLogic() {
            output = new VariableSet();
        }

        @Override
        public void initialize() {
            conversation.poke(DDE_ACTION, "start");
            conversation.poke("override", "100");
        }
        @Override
        public VariableSet computeNewOutput(VariableSet input) {
            String data = conversation.request(FuerzaMedia);
            output.add(outputInfo, Double.parseDouble(data));
            return output;
        }
        @Override
        public void free() {
            conversation.disconnect();
        }
        @Override
        public boolean isFinished(long acc) {
            if ((acc + cycleTime) > expTime) {
                return true;
            } else {
                return false;
            }
        }
    }
}

```

Listing 4: MicroProcess implementation.

```
public class SingleLoopMode extends Mode {
    private static final String NAME = "SINGLE LOOP";

    public SingleLoopMode() {
        super(NAME);

        // Creating the nodes
        OrganizationNode n1 = OrganizationNode.createModelNode(ModelType.
    SINGLE_LOOP);
        OrganizationNode nP = OrganizationNode.createProccesNode();
        OrganizationNode nR = OrganizationNode.createRefNode();
        // Adding the inputs
        n1.addInput(nR, 1);
        n1.addInput(nP, -1);
        nP.addInput(n1, 1);
        // Defining the layers
        newLayer();
        addNodeToLayer(nR);
        newLayer();
        addNodeToLayer(n1);
        newLayer();
        addNodeToLayer(nP);
    }
}

```

Listing 5: Single loop mode implementation.

```
public class AnticipationMode extends Mode {
    private static final String NAME = "ANTICIPATION";

    public AnticipationMode() {
        super(NAME);

        // Creating the nodes
        OrganizationNode n1 = OrganizationNode.createModelNode(ModelType.
    ANTICIPATION);
        OrganizationNode nP = OrganizationNode.createProccesNode();
        OrganizationNode nR = OrganizationNode.createRefNode();
        // Adding the inputs
        n1.addInput(nR, 1);
        nP.addInput(n1, 1);
        // Defining the layers
        newLayer();
        addNodeToLayer(nR);
        newLayer();
        addNodeToLayer(n1);
        newLayer();
        addNodeToLayer(nP);
    }
}

```

Listing 6: Anticipation mode implementation.

```
1  public class AnticipationMirroringMode extends Mode {
2      private static final String NAME = "ANTICIPATION+MIRRORING";
3
4      public AnticipationMirroringMode() {
5          super(NAME);
6
7          // Creating the nodes
8          OrganizationNode n1 = OrganizationNode.createModelNode(ModelType.
       ANTICIPATION);
9          OrganizationNode n2 = OrganizationNode.createModelNode(ModelType.
       MIRRORING);
10         OrganizationNode nP = OrganizationNode.createProccesNode();
11         OrganizationNode nR = OrganizationNode.createRefNode();
12         // Adding the inputs
13         n1.addInput(nR, 1);
14         n1.addInput(nP, -1);
15         n1.addInput(n2, 1);
16         n2.addInput(n1, 1);
17         nP.addInput(n1, 1);
18         // Defining the layers
19         newLayer();
20         addNodeToLayer(nR);
21         newLayer();
22         addNodeToLayer(n1);
23         newLayer();
24         addNodeToLayer(nP);
25         addNodeToLayer(n2);
26     }
27 }
28
```

Listing 7: Anticipation+Mirroring mode implementation.

C Test environment

The deployment of the implementation of the architecture can be seen in figure 28. The Raspberry Pi (*Pi 1*) will run the cognitive part of the architecture, on the other hand the Raspberry Pi (*Pi 2*) will run the executive part. The *Process host* has the mission of retrieving the process output from the *KERN Evo* machine and sending them to the architecture via ZeroC Ice, as well as receiving the action control from the architecture and sending it to *KERN Evo* machine. As we can see in the figure 28 the communication between the *Process host* and the *KERN Evo* is done via *Ethernet*. The *Registry* host will permanently run the *Icegrid registry* program to enable the discovering between hosts. Finally, the *Client* host can use the developed graphical user interface to interact with the different components.

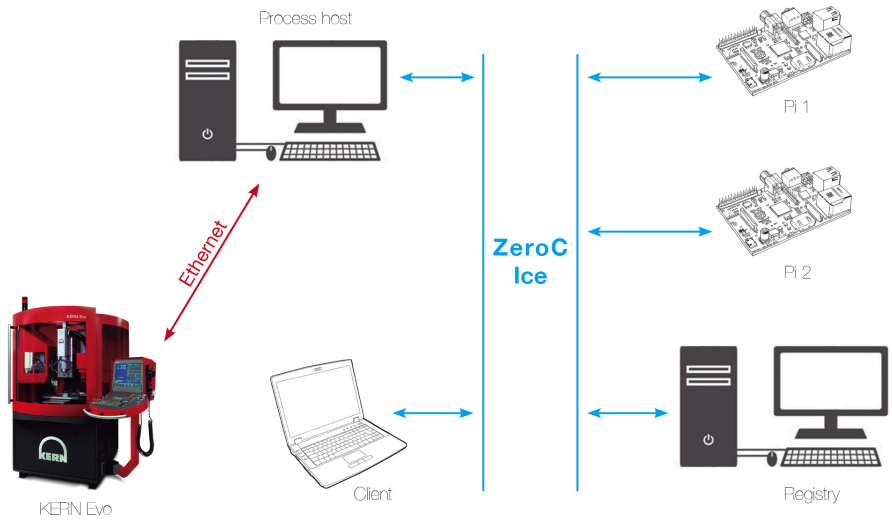

Figure 28: Deployment schema.

The whole real environment is presented in figure 29.

Figure 29: Whole architecture deployment

It is important to note that ZeroC Ice version 3.4 and Java version 1.7 have been installed in all the machines. *Client* host does not require the installation of Ice, it is sufficient with the Ice.jar because it has not to run a server. In order to have a greater understanding about the features of the deployment components, we will summarized them in following pages.

Process host

This machine has the following features:

Operating System: Microsoft Windows XP Profesional (version 2002), Service Pack 3.

CPU: Intel(R) Pentium(R) 4 CPU 3.20Ghz 3.19Ghz.

RAM: 2GB.

It is showed in figure 30.

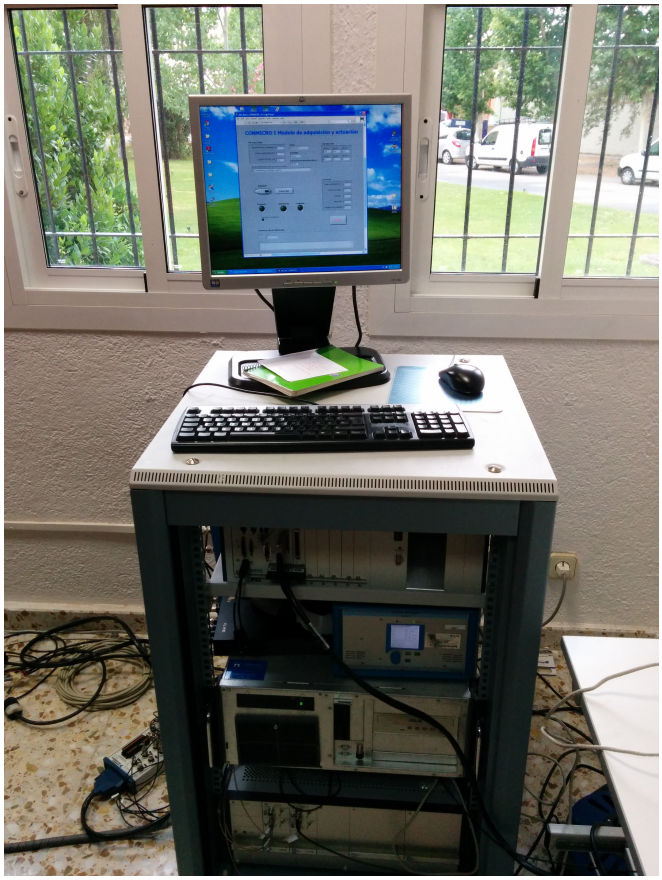

Figure 30: Process host features

Raspberry Pi

These machine have the following features:

Operating System: Raspbian.

CPU: ARM 1176JZFS a 700 MHz.

RAM: 512MB.

For further information see [25]. A close sight of this machine is presented in figure 31.

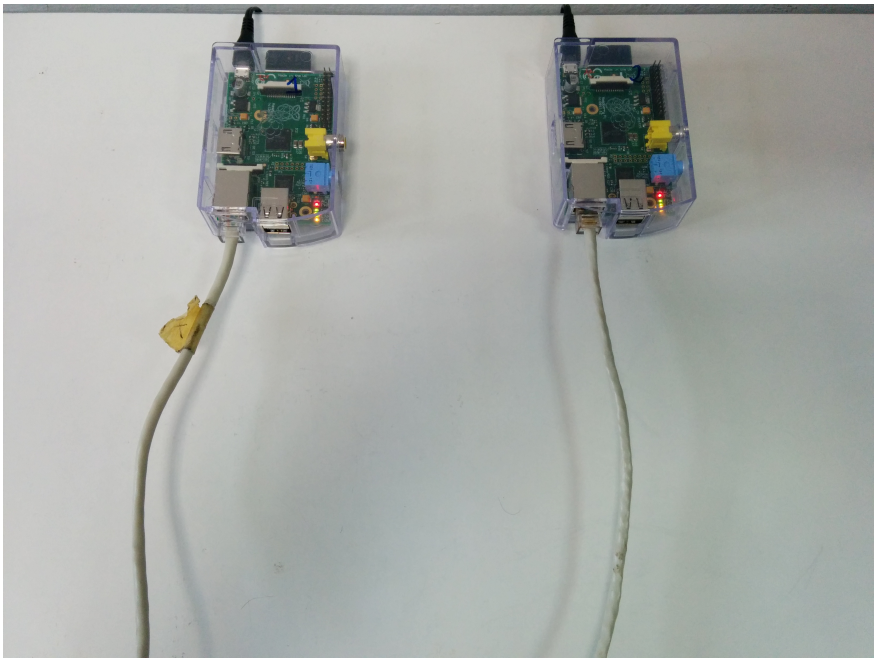

Figure 31: Raspberry Pi features

Registry host

This machine has the following features:

Operating System: Ubuntu 12.04 LTS.

CPU: Intel(R) Core(TMT) 2 CPU 6400, 2.13Ghz 2.13Ghz.

RAM: 2GB.

It is presented in figure 32.

Figure 32: Registry host features

KERN Evo

The digital direct feed drives fitted to the KERN Evo ultra precision machining centre provide fast acceleration and feed rates. These forces are absorbed by the polymer concrete monobloc machine frame.

The KERN Evo is specially designed for applications requiring the following features:

- Highest precision on the workpiece (deviation of position $P_a \pm 0.5\mu m$ according to VDI/DGQ 3441)
- Excellent surface quality $Ra < -0.1\mu m$
- Milling of critically machinable materials and hardened steel
- High productivity
- High acceleration rates
- High feed rates
- Automatic workpiece loading for batch production (available for 3 and 5 axes machining)

The spindle has the following features: 500-50000 rpm, permanently grease lubricated, vector-controlled and 1.5Nm with 6.4Kw. For further information see [26].

An image of the machine is presented in figure 33.

Figure 33: Kern Evo features

Client

The client machine can be recipient of any hardware configuration, the only important issue that it has to have is Ice (version 3.4) and Java (version 1.7) installed. A Graphical User Interface (GUI) application has been developed in order to ease the task of establishing the communication between the process and the control application. The GUI application is very intuitive, when it runs a window with three sections is displayed (figure 34). In this window a connection with the process can be done uniquely introducing the process's name (figure 35). Once the communication with the process is established the process's parameters can be set (figure 36) and the control application

can be initialized (figure 37). The next step is selecting the cognitive modes of the application and the default mode (figure 38). It is important to note that it is possible to enable the learning and also do an optimization of the model's parameters before the control. Once these steps are done it is able to indicate that we want to control the process (figure 39)and then we can start the process (figure 40). The output's process appears in the right section (figure 41). Once the process is ended a graphical plot can be shown (figure 42) and it is possible to save the results in a file (figure 43).

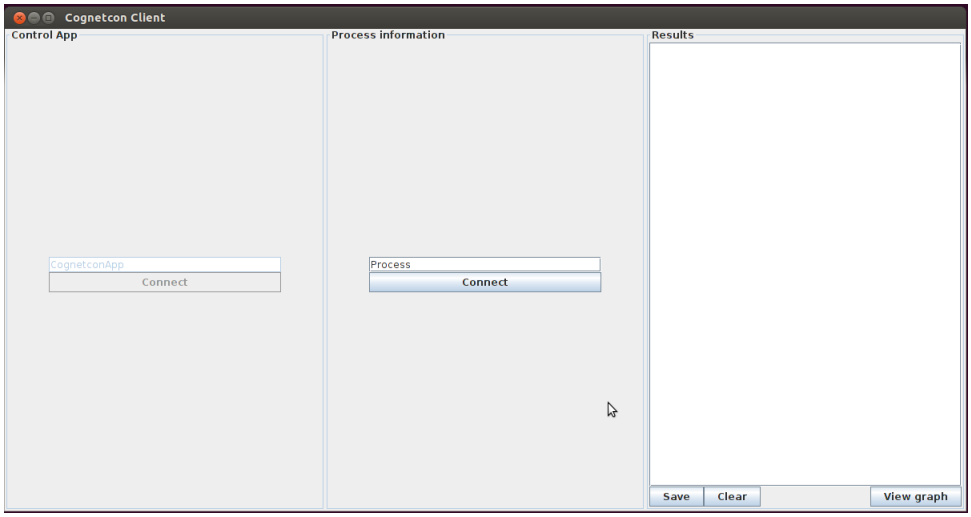

Figure 34: GUI application: first window.

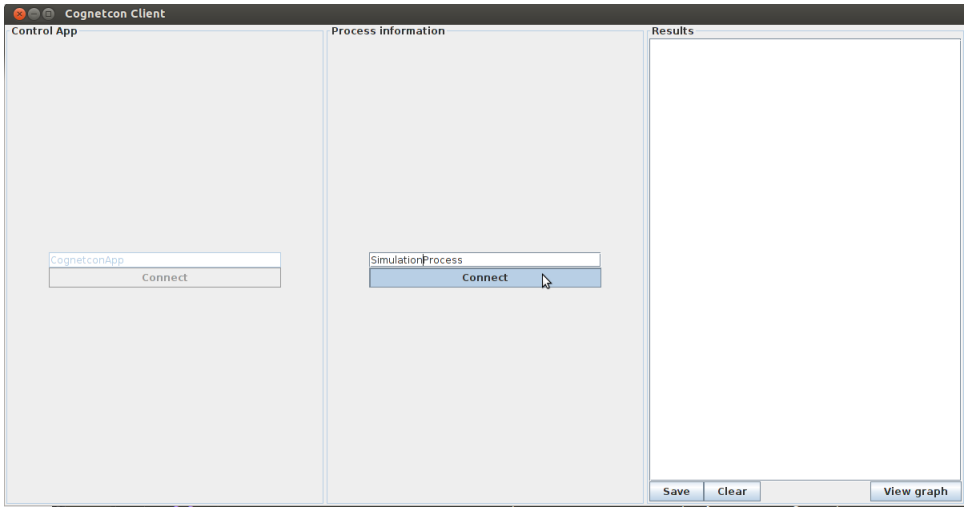

Figure 35: GUI application: connection with the process.

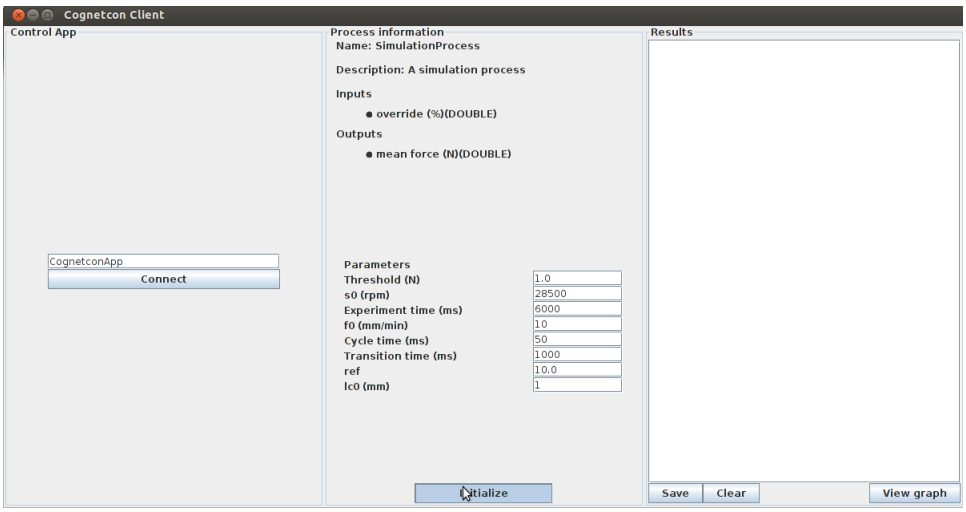

Figure 36: GUI application: process configuration.

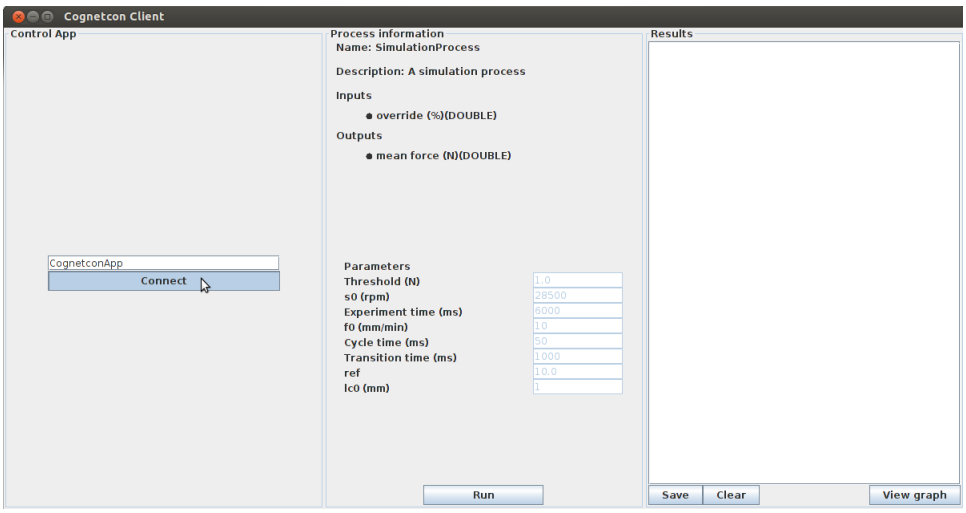

Figure 37: GUI application: connection with the control application.

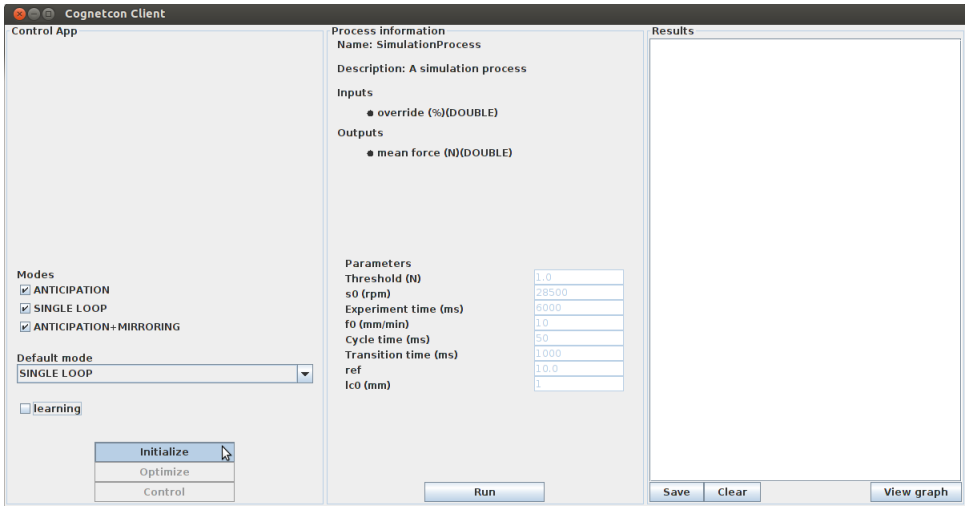

Figure 38: GUI application: selecting cognitive modes.

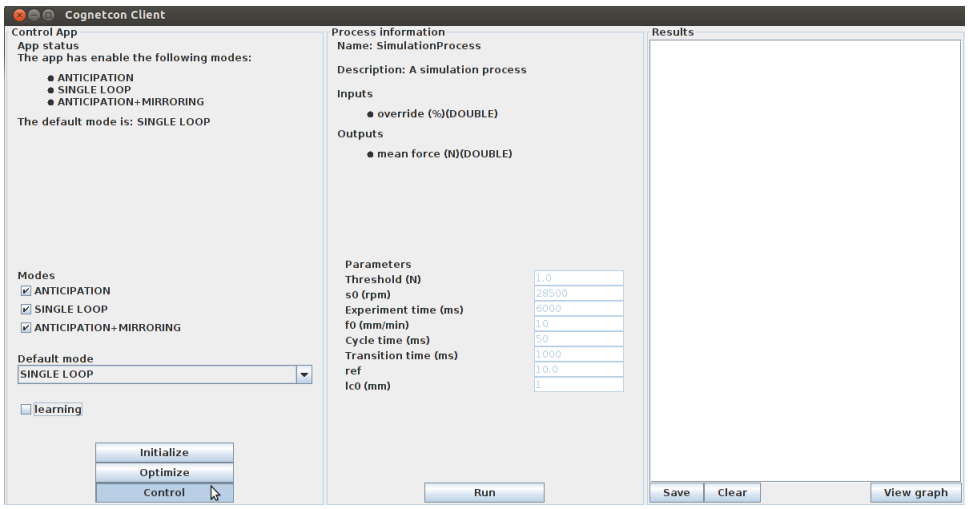

Figure 39: GUI application: indicating we want to control.

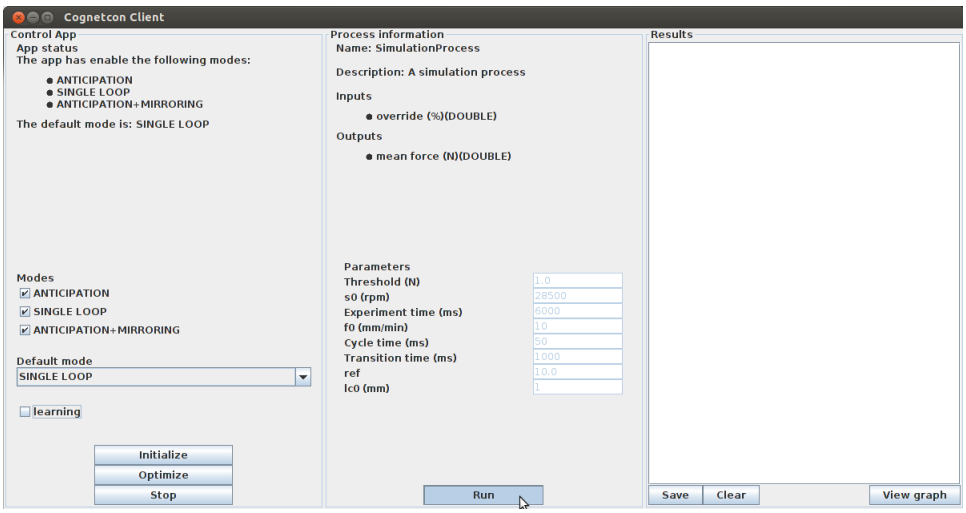

Figure 40: GUI application: starting the process.

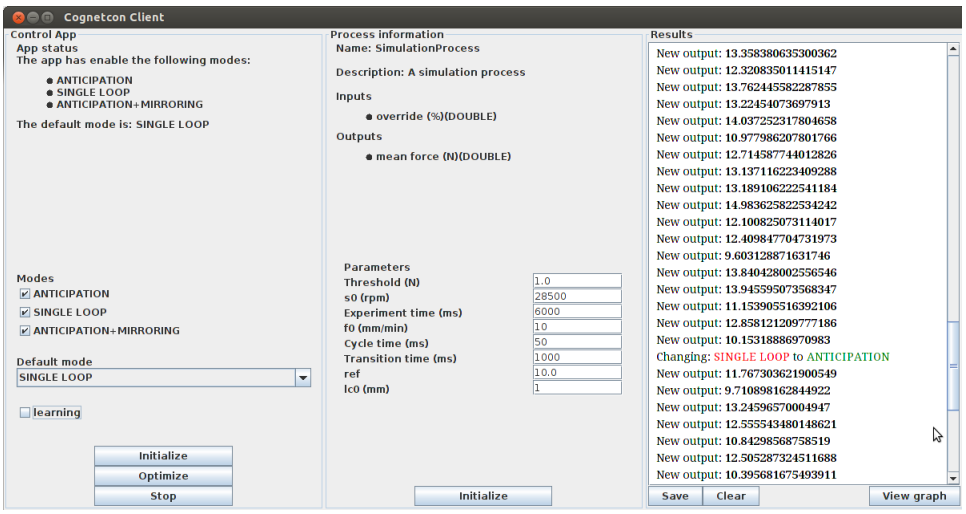

Figure 41: GUI application: process output.

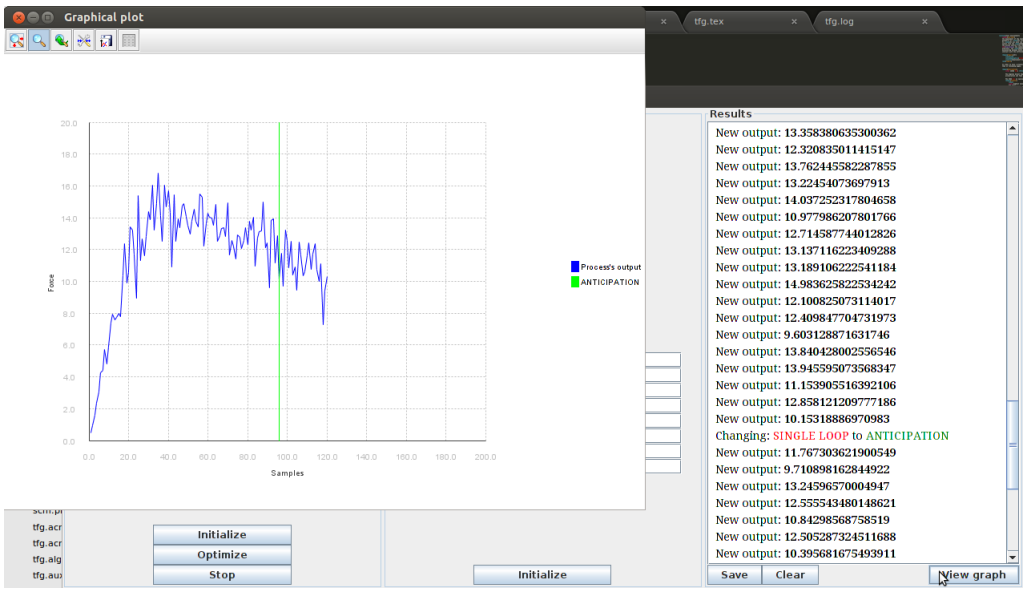

Figure 42: GUI application: graphical plot.

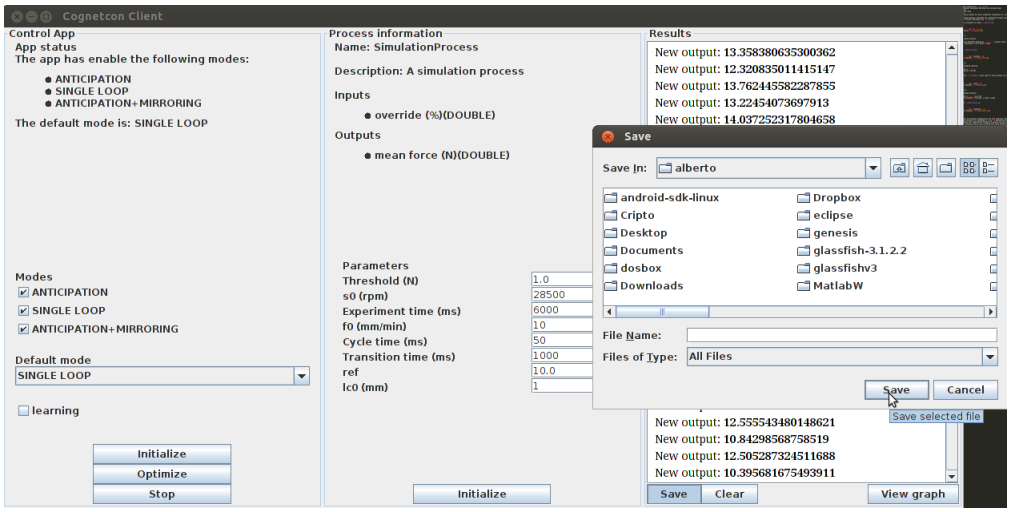

Figure 43: GUI application: saving results.